MASTER EN SCIENCES INFORMATIQUES

Audit de migrations de Build System dans l'écosystème d'Eclipse

Bollen, Mathieu

Award date:
2013

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2012–2013

Audit de migrations de Build System dans l'écosystème d'Eclipse

Mathieu BOLLEN

Maître de stage : Bram ADAMS

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Naji HABRA

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé et mots clés

Résumé : Tout comme le code source, le Build System utilisé pour la construction d'un projet évolue, ce qui peut mener à un Build System tellement complexe qu'il prend trop de temps et de travail à maintenir.
Une solution est de migrer vers une technologie de Build System plus puissante. L'étude de la migration du Build System de Linux et KDE, deux projets en langage C, a montré que celle-ci peut être risquée, longue et complexe.
Nous avons mené une étude de cas de sept projets Java open source qui ont réalisé une migration au cours de ces dernières années, abandonnant la technologie Ant pour utiliser Maven.
Notre objectif était de mieux comprendre quelles sont les raisons qui motivent une migration, quelle est la complexité de la migration et dans quelle mesure la migration a un impact positif sur la qualité du Build System.
Les résultats ont montré que les principales motivations sont la complexité de l'ancien Build System et les avantages qu'offre la nouvelle technologie.
L'utilisation de l'outil Tycho permet aux développeurs de réaliser leur migration très rapidement et facilement.
Bien que l'impact sur l'effort de maintenance n'a pas été significatif sur les projets analysés, l'amélioration est malgré tout bien présente et proportionnellement à la facilité de migration, elle reste avantageuse.

Abstract : As the source code, the Build System used for the construction of a project evolves, and that can lead to a Build System so complex that it takes too much time and work to maintain.
A way out is to migrate toward a more powerful Build System technology. The study of the migration of Linux and KDE, two projects written in C, showed that it can be risky, long and complex.
We conducted a case study of seven Java open source projects that performed a migration in the recent years, leaving the Ant technology to use Maven.
Our goal was to have a better understanding of the reasons that motivate the migration, what is the complexity of the migration and to what extent the migration has a positive impact on maintenance effort.
The results showed that the main reasons are the complexity of the previous Build System and the advantages of the new technology.
The use of the tool Tycho helps the developers to perform their migration quickly and easily.
Even if the impact on the maintenance has not been significant, the improvement is still present and in proportion to the ease of migration, it is advantageous.

Mots clés : Build System, migration, étude de cas, qualité, évolution

Avant-propos

Le travail décrit dans ce mémoire est le résultat de trois mois de stage à l'École Polytechnique de Montréal, au Canada. J'y ai intégré l'équipe MCIS, "*Maintenance, Construction and Intelligence of Software*" [1], supervisé par le Dr Bram Adams.

J'aimerais remercier le Dr Naji Habra de m'avoir donné l'opportunité de faire ce stage, et pour sa supervision. J'aimerais également remercier le Dr Bram Adams, pour sa supervision durant le stage et son aide lors de la rédaction du mémoire.

Finalement, j'aimerais remercier mes parents pour leur soutien et leur aide tout au long de mes études. En particulier, j'aimerais remercier ma maman pour le temps qu'elle m'a accordé pour relire mon mémoire.

1. http ://mcis.polymtl.ca/index.shtml

Table des matières

Table des figures

CHAPITRE 1

Introduction

Le Build System est l'application qui automatise la création d'un logiciel. Il interagit directement ou indirectement avec la plupart des stakeholders du projet logiciel.
L'augmentation constante de la taille et la complexité du Build System a motivé de nombreux projets à migrer leur Build System vers une nouvelle technologie pour en réduire notamment la complexité. Cependant quelques études récentes ont démontré que le processus de migration pour les projets C peut être long et difficile. Ce domaine reste encore largement ignoré des recherches actuelles.

Dans ce travail, nous allons réaliser une étude de cas sur sept projets Java open source de l'écosystème d'Eclipse, pour essayer de mieux comprendre les raisons qui motivent une migration, en cerner les difficultés et les effets réels.

La première partie présentera la problématique qui nous occupe, en précisant le contexte de l'étude et les recherches qui ont été effectuées sur le sujet.
Dans la seconde, nous présenterons le contexte et les projets de notre étude de cas.
Dans la troisième, nous expliquerons la méthodologie que nous avons appliquée pour mener à bien notre étude de cas.
Dans la quatrième partie, nous présenterons les résultats de l'étude, et nous les analyserons.
Enfin, nous ferons une évaluation de la méthodologie utilisée.

CHAPITRE 2

Problématique

2.1 Contexte

Le Build System est une application qui automatise la création d'un logiciel. Lors de son exécution, le Build System effectue en général les tâches suivantes :

- compiler tout le code source en code binaire
- effectuer les tests de vérification du bon fonctionnement du logiciel
- rassembler le code binaire en packages
- déploier le logiciel dans le système de production
- créer la documentation et/ou les release notes.

L'utilisation d'un Build System est essentielle lorsque le projet prend de l'ampleur.
Quand le projet utilise plusieurs modules de code source, le nombre d'opérations à effectuer pour construire le programme devient de plus en plus élevé, et il est alors impératif de suivre un ordre particulier dans les opérations. En effet, certains modules peuvent avoir en pré-requis d'autres modules, qui devront être compilés avant eux. Si la compilation est toujours faite manuellement, il n'est pas facile de garantir le respect de cet ordre et de ne pas oublier de module.

Le Build System a donc un rôle essentiel dans le développement d'un projet. Il simplifie le travail des développeurs, qui doivent reconstruire un prototype testable du projet après avoir modifié le code source.
C'est également un élément clé dans la coordination de l'équipe. Par exemple, la méthodologie de développement d'*intégration continue*[1] requière une exécution automatique des constructions du projet et une publication des résultats par email pour apporter aux développeurs un feedback sur la qualité du programme.

Pour les projets développés en Java, les deux technologies de Build System majeures actuelles sont Ant et Maven.

Ant, un acronyme pour "Another Neat Tool", signifiant "Un autre outil soigné", a été créé en 1999 par James Duncan Davidson. L'idée de Ant était de corriger les défauts principaux de *make*, la technologie standard des projets C/C++. Son principal défaut était sa forte dépendance au système d'exploitation sur lequel était exécuté le Build System, due à l'utilisation de scripts Shell, utilisable uniquement dans les environnements Linux.
Ant a donc été designé pour être petit, extensible, et indépendant du système d'exploitation. Cette dernière caractéristique garantissait ainsi au langage Java une portabilité complète.

Le code de Ant est rédigé en XML, dans des fichiers typiquement nommés "*build.xml*".
Voici un exemple simple de fichier *build.xml*[2]. Ce fichier définit quatre *targets*, qui seront les tâches à effectuer, nommées *"init", "compile", "dist"* et *"clean"*. Le mot-clé "*depends*" indique que la target ne peut être effectuée que lorsque la target dont elle dépend a été terminée. Par exemple, pour éxécuter la target *"dist"*, qui va créer le répertoire de distribution, il faut que la target *"compile"* qui compile le code source ait déjà été exécutée.
Les balises internes indiquent les actions à réaliser lors de l'éxécution de la tâche. Par exemple, la target *"clean"* va éxécuter les commandes *delete dir="$build"/* et *delete dir="$build"/*, permettant de supprimer les deux dossiers créés lors de l'exécution du Build System.

1. Pratique de développement logiciel où les membres d'une équipe intègrent leur travail quotidiennement - http ://martinfowler.com/articles/continuousIntegration.html
2. Source : http ://ant.apache.org/manual/using.html

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Quant à Maven, sa première version a été mise à disposition des développeurs en juin 2007, également par la fondation Apache. Maven a été créé avec l'idée de standardiser le processus de build, suite à l'observation que de nombreux projets Java réimplémentent systématiquement les mêmes targets Ant.

Le paradigme utilisé par Maven est *Project Object Model (POM)*. Il décrit le projet, ses dépendances avec des modules externes et l'ordre de construction à suivre. Ce POM se matérialise par des fichiers *pom.xml*. Le fichier principal est positionné à la racine du projet, il est appelé POM parent. Les fichiers POM sont organisés en hiérarchie, ce qui permet l'héritage des propriétés.

L'approche de standardisation de Maven diffère totalement de celle de Ant. Là où Ant laisse les développeurs libres dans l'architecture de leur projet et dans l'écriture du Build System, Maven utilise le concept de *convention plutôt que configuration*. Cela signifie que Maven impose une structure standardisée du projet ainsi qu'une nomenclature spécifique des dossiers. Tant que les développeurs suivent ces conventions, Maven peut utiliser sa configuration par défaut. Si les développeurs ont besoin de s'écarter de la convention, Maven l'autorise mais nécessite que les développeurs le précisent dans la configuration du projet.

Maven utilise un *cycle de vie* de construction. Lorsque la construction du projet commence, Maven va réaliser automatiquement toutes les étapes du build, de façon séquentielle. Cela signifie qu'une étape ne sera pas réalisée tant que l'étape précédente n'est pas terminée. Les phases par défaut du cycle de vie sont :

- *validate* - vérifie que le projet est correct et que toutes les informations nécessaires sont disponibles
- *compile* - compile le code source du projet
- *test* - teste le code source compilé. Ces tests ne doivent pas nécessiter que le code soit en package ou déployé.
- *package* - package le code compilé dans son format de distribution, par exemple un JAR.
- *integration-test* - effectue les tests d'intégration du package

- *verify* - effectue tous les contrôles pour vérifier que le package est valide et réponde aux critères de qualité.
- *install* - installe le package dans le répertoire local, pour l'utiliser comme dépendance dans d'autres projets locaux.
- *deploy* - Copie, dans un environnement d'intégration ou de release, le package final dans le répertoire distant pour le partager avec les autres développeurs et projets.

Il est possible d'ajouter d'autres phases au cycle de vie, mais elles doivent alors être configurées dans le POM parent.

Pour simplifier encore le travail des développeurs, Maven s'occupe aussi automatiquement des sous-dépendances des plug-ins du Build System. Lorsqu'un plug-in est ajouté au Build System, et qu'il dépend de différents autres plug-ins qui ne sont pas encore intégrés au Build System, Maven va alors les rechercher et les télécharger automatiquement. Il en va de même pour le téléchargement de leurs mises à jour.

Voici un exemple simple de fichier *pom.xml*[3]. La différence notable avec le build.xml est l'absence de configuration manuelle. Ant réclamait de construire des targets, avec leurs actions à réaliser et leurs interdépendances explicitement écrites. Grâce au cycle de vie, apporté par le concept de convention, le pom.xml n'a pas besoin de ces informations.
Les balises *"dependencies"* listent les plug-ins dont le Build System du projet dépend. Dans notre exemple, le projet utilise le plug-in JUnit, à la version 3.8.1.

3. Source : http ://maven.apache.org/guides/getting-started/

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

2.2 État de l'art

2.2.1 Évolution du Build System

Utiliser un Build System ne se limite pas à simplement le rédiger. Il a aussi besoin d'être maintenu pour pouvoir suivre les évolutions que subit le projet au cours de son développement. Cette maintenance peut impliquer l'addition de règles de build pour accueillir de nouveaux modules de code source, ou des ajustements à la configuration du compilateur.

Cette maintenance demande un effort supplémentaire non-trivial de la part des développeurs.

De nombreuses études sur l'évolution des Build Systems ont été menées dans le but de trouver des patterns récurrents dans leur évolution, et d'évaluer la charge de travail que la maintenance impose aux équipes de développement.

En 2002, Kumfert *et al.*[3] ont mené une enquête auprès de 34 développeurs venant de *Lawrence Livermore National Labs* pour évaluer la surcharge de travail perçue par la maintenance du Build System.
Leurs résultats étaient qu'en moyenne cette surcharge était de 12% de l'effort de développement, c'est à dire que 12% du développement est dédié à la maintenance du Build System.

Une autre étude, menée par Adams *et al.*[1] s'est intéressé à l'évolution d'un Build System largement utilisé, celui du kernel du célèbre système d'exploitation Linux.
Ils ont observé que l'évolution d'un projet ne passe pas uniquement par celle du code source. À chaque fois que du code source est ajouté, ou que des modules existants sont déplacés, le Build System doit être modifié pour que le projet reste compilable. L'évolution habituelle du code source a donc un impact direct sur le Build System, mais l'inverse est également vrai.
Leur objectif est de quantifier ces observations, en analysant l'évolution du code source et du Build System, de leur création jusqu'aux dernières versions disponibles.

Leur première observation est que le Build System évolue bien au fil du temps, et suit la même tendance que l'évolution du code source.
La deuxième observation est que la complexité du Build System augmente aussi avec le temps. Le nombre de dépendances entre les targets grandit, et avec elle la difficulté de comprendre les relations entre elles. Ces difficultés augmentent le temps nécessaire pour comprendre et modifier le Build System, et constituent une source potentielle d'erreurs de build.
La troisième observation montre que l'évolution du Build System est dirigée par sa maintenance. Les développeurs doivent fournir des efforts spécifiques pour gérer cette complexité croissante.

Le kernel de Linux est développé en C, en utilisant la technologie Make pour son Build System. McIntosh *et al.*[4] ont voulu vérifier si ces observations réalisées sont confirmées avec d'autres technologies. Ils ont alors étudié quatre projets open source développés en Java et utilisant la technologie de Build System Ant.

Ils ont eux aussi analysé l'évolution des Build Systems, pour vérifier si la taille et la complexité du Build System évoluent, et si elles sont corrélées à l'évolution du code source.
Adams *et al.* ont observé que (1) le Build System de Linux évolue, (2) la complexité du Build System augmente au fil du temps, et (3) la maintenance dirige l'évolution du Build System.
Les deux premières observations ont été confirmées dans cette étude. Les Build Systems Ant augmentent en taille et en complexité au fil du temps, à moins qu'un effort explicite ne soit fourni pour les restructurer. Cette évolution est, elle aussi, corrélée à l'évolution du code source.
Pour la troisième observation, McIntosh *et al.* ont remarqué que les raisons qui motivent l'évolution du Build System diffèrent. Par exemple, les développeurs du Build System Make doivent fournir beaucoup d'efforts pour conserver un Build System modulaire, alors que Ant et Maven leurs fournissent des outils spécifiques pour garantir la modularité.

Leur étude a démontré que les Build Systems des projets Java et C évoluent de manière similaire. De nombreuses motivations de l'évolution des Build Systems des projets C sont présents dans les Build Systems des projets Java, mais il y a malgré tout d'importantes différences.

Plus tard, Adams *et al.* ont étendu cette étude aux Build Systems Maven, où ils ont obtenu les mêmes résultats que pour les Build Systems Ant[5].

L'étape suivante dans l'analyse de l'évolution des Build Systems a été d'analyser plus en détail cette surcharge de travail qu'impose la maintenance du Build System. Pour cela, McIntosh *et al.*[6] ont mené une étude empirique détaillée sur dix larges projets C et Java.

Leur étude est divisée en trois dimensions :

1. La taille et le taux de changement du Build System
2. Le couplage du Build System avec le code source et avec le code de test
3. La répartition de l'implication des développeurs dans la maintenance du Build System

Les résultats de la première dimension montrent que le Build System est relativement petit en taille, composant 9% (médiane) des fichiers des projets étudiés. Par contre, le Build System change fréquemment, à un taux proche de celui du code source.

Les analyses du couplage se basent sur les *work items*, autrement dit le groupe de révisions modifiant le même objet, comme une correction d'un même bug.
Les résultats montrent que 4 à 16% des *work items* du code source et 8 à 20% des *work items* du code de test des projets Java analysés requièrent un changement dans le Build System, de même que 27% des *work items* du code source et 44% des *work items* du code de test des projets C analysés.

Les projets analysés ont deux styles d'implication dans le Build System ; un style concentré, où la majorité des modifications sont réalisées par une petite équipe d'experts du Build System, et un style dispersé, où la plupart des développeurs contribuent aux modifications du Build System.
Jusqu'à 79% des développeurs du code source et 89% des développeurs du code de test sont significativement implicé dans la maintenance du Build System, mais l'utilisation d'une équipe d'experts en Build System peut réduire cette proportion de développeurs impliqués à 22% des développeurs du code source et 24% des développeurs du code de test.

2.2.2 Migration du Build System

En évoluant, le Build System peut atteindre un stade où la complexité devient tellement importante qu'elle impacte drastiquement la productivité des développeurs, et ralentit le développement du projet. C'est ce qu'il s'est produit pour le projet KDE[7]. KDE 3 utilise la technologie *autotool* pour construire son projet. Cette technologie est vraiment difficile à utiliser, au point que les développeurs la surnomment "auto-hell" !
Pour la nouvelle version, KDE 4, l'équipe de développement a décidé de laisser de coté autotool et de migrer le Build System dans le but le restructurer et d'utiliser une technologie plus facile à manipuler.

La première tentative des développeurs a été de migrer vers SCons, une technologie de Build System basée sur Python. Très vite, plusieurs défauts majeurs de SCons sont apparus, notamment une couche "configuration" immature et des difficultés de portage vers d'autres plateformes. La migration a donc été arrêtée, et les développeurs se sont tournés vers CMake, une version cross-plateforme de Make. La deuxième tentative de migration a cette fois été un succès.

Linux a lui aussi entrepris une migration du Build System de son kernel, entre les versions 2.4 et 2.6.
Les développeurs ont tenté une première migration pour la version 2.5. L'objectif était de réécrire la couche "configuration" avec un langage plus concis (CML2) et d'adopter une technologie Make non-récurcive pour la couche "construction".
Malheureusement, la migration fut un échec par manque de rétro-compatibilité de CML2 et de plan de migration incrémental pour le Make non-récurcif.

La deuxième tentative de migration pour la version 2.6 fut quant à elle un succès. Ils ont choisi de migrer la couche "configuration" vers KConfig, et de restructurer de façon incrémentale la couche "construction" vers une architecture plus puissante, toujours basée sur la technologie récurcive de Make.

Adams *et al.*[2] ont étudié les deux tentatives de migration de Build System échouées et les deux réussies, pour ces deux projets open source. L'objectif est d'identifier une méthodologie commune pour la migration du Build System et les challenges majeurs associés à la migration.

Ils ont découvert que le processus de migration de Build System ressemblait au modèle en spirale utilisé dans le développement du code source, et ils ont isolé des phases individuelles de ce processus : analyse des risques et planification, implémentation et évaluation.

Ils ont également découvert quatre challenges majeurs qui vont attendre les développeurs qui entreprendront une migration de leur Build System :

1. le rassemblement des exigences,
2. les problèmes de communication avec les différentes stakeholders,
3. l'équilibrage de l'amélioration des performances avec la complexité du code du Build System,
4. l'évaluation efficace des prototypes de Build System.

Adams *et al.* ont également observé qu'aucune des équipes de développement de KDE et de Linux n'a correctement réuni les exigences de leur nouveau Build System lors de leur première tentative. Ceci était dû à une mauvaise implication des stakeholders et un manque de communication avec les experts du Build System.
Les développeurs de ces deux projets ont compris leurs erreurs, ce qui leur a permis de réussir leur deuxième migration.

2.3 Questions de recherche

Lors de l'analyse des migrations de Linux et de KDE, nous avons observé que celles ci étaient des opérations risquées, longues et compliquée. Nous voulons donc étendre cette analyse de migrations à d'autres technologies pour mieux en comprendre le processus.

Dans ce mémoire, nous allons mener une étude de cas de sept projets, pour tenter de répondre aux trois questions de recherche qui nous préoccupent :

- RQ1 : *"Quelles sont les raisons qui motivent une migration ?"*
- RQ2 : *"Quelle est la complexité de la migration ?"*
- RQ3 : *"Dans quelle mesure la migration a un impact positif sur la qualité du Build System ?"*

CHAPITRE 3

Etude de cas

Dans ce troisème chapitre, nous allons dans un premier temps décrire le contexte de notre étude de cas. Ensuite, nous présenterons les différents projets qui la composent.

3.1 Contexte de l'étude de cas

Dans cette étude, nous nous intéresserons exclusivement à des projets appartenant à l'écosystème d'Eclipse [1]. Eclipse est l'un des environnements de développement les plus populaires au monde, constituant la base de la plupart des outils de développement Java sur le marché.

Les raisons du choix de l'écosystème d'Eclipse pour notre étude de cas incluent le statut important d'Eclipse dans le monde du développement open source et la migration massive des Build Systems à travers les projets d'Eclipse.

Auparavant, ces projets utilisaient la technologie Ant pour le développement de leur Build System. Les uns après les autres, ils ont décidé de migrer vers la technologie Maven. Cette dernière propose en effet de nombreux avantages, tels que [2]

1. http ://www.eclipse.org
2. Source : http ://www.kblaney.com/benefits-of-maven-over-ant/

- **La standardisation de la structure du Build System**
  Tous les projets Maven possèdent une même structure standardisée, ce qui permet une compréhension beaucoup plus aisée de chaque projet. En contrepartie, cette standardisation contraint les développeurs à respecter la structure et la nomenclature des dossiers.

- **La gestion des dépendances**
  Avec Maven, les dépendances transitives sont gérées automatiquement par le Build System lui-même. Lorsque les développeurs ajoutent une dépendance à une bibliothèque, ils ne doivent pas se soucier des dépendances propres à cette bibliothèque. Maven s'en chargera, en allant lui-même chercher les éléments dont la bibliothèque a besoin.

- **Des builds divisés en composants**
  Les builds sont divisés en composants plus petits, qui permettent de tester et d'intégrer très rapidement des changements de code.

- **La réduction de la duplication**
  Maven peut utiliser une hiérarchie de ses POM pour réduire la duplication qui existe typiquement dans les projets Ant. Par exemple, le POM parent peut prendre en charge tous les détails d'une configuration, et tous les POM du projets peuvent hériter de cette configuration.

- **L'utilisation d'autres langages**
  Maven possède un outil lui permettant d'utiliser du code provenant d'autres technologie, comme par exemple du code Ant. Cet outil permet de pallier à d'éventuelles fonctionnalités que Maven ne possède pas.

- **La simplicité d'exécution**
  Grâce aux cycles de vie de Maven et l'utilisation d'une structure de projet standardisée, tous les Build Systems utilisant Maven n'ont besoin que d'un simple "mvn clean install" pour être exécutés.
  Par contre Ant n'a pas de méthode unifiée pour éxécuter le Build System, ce qui peut être un frein dans les communautés open source, où les contributeurs potentiels pourraient être démotivés s'ils ont de trop grandes difficultés à construire le projet.

Jusqu'il y a quelques années, il était impossible d'utiliser Maven au sein de l'écosystème Eclipse à cause d'une incompatibilité avec le fonctionnement d'OSGi [3], le framework de gestion des plug-ins d'Eclipse.
OSGi a ses propres méta-données pour exprimer les dépendances, la localisation des fichiers sources, etc.
Ces méta-données sont généralement trouvées dans le fichier POM de Maven.
Pour contourner cette incompatibilité, la communauté Eclipse a développé Tycho [4], un plug-in d'Eclipse composé d'un ensemble de plug-ins Maven, qui sert d'intermédiaire entre OSGi et Maven.

Tycho va utiliser les méta-données natives des plug-ins Eclipse et des Bundles OSGi. Il va également utiliser le POM pour configurer et diriger le build. Il permet ainsi à Maven d'utiliser OSGi et les méta-données d'Eclipse pendant la construction du projet.
Tycho utilise les méta-données et les règles de OSGi pour résoudre les dépendances du projet dynamiquement et les injecter dans Maven pendant l'exécution du build.

L'arrivée à maturité de Tycho a convaincu les développeurs de nombreux projets à migrer leur Build System vers Maven, tout en restant dans l'écosystème d'Eclipse.

3. Groupe de classes java qui permettent d'offrir des fonctionnalités supplémentaire au code l'utilisant.
4. http ://eclipse.org/tycho/

3.2 Présentation des projets analysés

Nous avons choisi d'analyser les sept projets suivants : LinuxTools, CDT, BPEL, JDT, Mylyn, PDE et JBossTools.
Nous avons sélectionné des projets de toutes tailles, quatre d'entre eux (BPEL, JDT, Mylyn et PDE) sont de petite taille, alors que LinuxTools et CDT sont de taille plus importante, et JBossTools est quant à lui d'une taille considérable.
Tous ces projets ont migré de la technologie Ant vers la technologie Maven il y a quelques années. Nous avons utilisé l'intégralité de leur historique, qui remonte pour le plus ancien à juin 2001, et pour le plus récent à février 2007.

Nous allons ici faire une courte présentation de ces projets.

LinuxTools

Le projet LinuxTools [5] vise à apporter un IDE [6] C et C++ complet aux développeurs Linux. Il s'appuie sur les fonctionnalités d'édition de code source et de debugging de CDT et intègre des outils de développement natifs populaires comme Valgrind, OProfile, RPM, SystemTap, GCov, GProf, LTTng, etc.

CDT

Le projet CDT [7] fournit un IDE C et C++ totalement fonctionnel, basé sur la plateforme Eclipse. Les fonctionnalités de CDT incluent un éditeur (avec coloration de la syntaxe, et autocomplétion du code), un exécuteur, un debugger, un moteur de recherche et un générateur de Makefile.

5. http ://www.eclipse.org/linuxtools/
6. Integrated Development Environment
7. C/C++ Development Tooling - http ://www.eclipse.org/cdt/

BPEL Designer

L'objectif du projet BPEL [8] est d'ajouter un support complet à Eclipse pour la définition, la création, l'édition, le déploiement, les tests et le debugging des processus WS-BPEL 2.0.
WS-BPEL (Web Services Business Process Execution Language) est une spécification neutre développée par OASIS pour spécifier les processus business comme un ensemble d'intéractions entre les web services. En fournissant ces outils, ce projet vise à construire une communauté autour du support de BPEL dans Eclipse.

JDT

Le projet JDT [9] fournit l'outil plug-in qui implémente un IDE Java qui soutient le développement d'applications Java, y compris les plug-ins Eclipse. Il ajoute une nature de projet Java et une perspective Java au Workbench d'Eclipse, ainsi que de nombreuses vues, éditeurs, wizards, constructeurs, et outils de fusion et refactoring de code.

Ce projet est divisé en plusieurs composants dont le développement est indépendant, chacun ayant sa propre équipe de développement. Il n'existe donc pas un répertoire GIT unifié dans lequel nous pouvons observé l'évolution du Build System de l'ensemble du projet, chaque répertoire ayant son propre Build System. Nous avons donc sélectionné les deux composants dont le développement est le plus actif.
Le premier est JDT-Core, qui contient la structure principale du projet.
Le deuxième composant est JDT-UI, qui est le composant qui s'occupe de développer l'interface utilisateur.

Mylyn

Mylyn [10] fournit au développeur une interface spécifique aux tâches sur lesquelles il travaille. Mylyn surveille les activités de l'utilisateur et essaie d'identifier quelles sont les informations pertinentes à la tâche en cours. L'objectif est d'améliorer la productivité en réduisant la recherche et la navigation dans les fichiers.

8. Business Process Execution Language - http ://www.eclipse.org/bpel/
9. Java development tools - http ://www.eclipse.org/jdt/
10. http ://www.eclipse.org/mylyn/

PDE

Le projet PDE [11] fournit des outils pour créer, développer, tester, debugger, construire et déployer des plug-ins Eclipse. PDE fournit aussi des outils complets pour OSGi, ce qui en fait un environnement idéal pour la programmation par composants, pas uniquement pour le développement de plug-in Eclipse.

JBossTools

JBoss Tools est un project-cadre pour un ensemble de plugins Eclipse et de fonctionnalités, conçu pour aider les développeurs de JBoss et J2EE à développer des applications. JBoss Tools supporte entre autres Hibernate, JBoss AS, Drools, jBPM, JSF, (X)HTML, Seam, Maven, JBoss ESB, JBoss Portal, etc.

Nous ne disposons de données que jusqu'à fin 2012, car l'équipe de développement a décidé d'abandonner le répertoire GIT principal pour le séparer en quatre parties distinctes.

JBoss Tools fut l'un des premiers projets à migrer son Build System vers Maven en utilisant Tycho, et ce bien avant que ce plug-in ne soit mature. Ils ont donc joué un rôle de pionnier, là où beaucoup de projets analysés dans cette étude ont attendu que Tycho soit mature et que des outils et des exemples soient disponibles pour les aider à réaliser la migration.

11. Plug-in Development Environment - http ://www.eclipse.org/pde/

CHAPITRE 4

Méthodologie

Pour répondre à nos questions de recherche, nous avons établi une méthodologie basée sur deux approches.

La première est une approche quantitative. Pour cette approche, nous analysons les données que nous tirons de l'intégralité de l'historique du répertoire GIT de chaque projet. GIT est un système de contrôle de version et de gestion du code source, dans lequel chaque développeur ajoute les modifications qu'il a apportées au projet via des *commits*.
L'analyse de ces commits permet d'évaluer la qualité du Build System du projet, grâce à une série de dimensions que nous avons établies. Les dimensions sont des aspects du projet qui permet d'évaluer un critère de qualité.
Le choix de ces dimensions s'est basé sur les analyses déjà réalisées par Adams *et al.*[1] et McIntosh *et al.*[5][6].

La deuxième approche est qualitative. Celle-ci nous permet d'avoir des informations complémentaires aux informations fournies par les données quantitatives. Ceci dans le but de mieux comprendre la migration du Build System. Nous avons donc demandé un feedback au développeurs qui ont travaillé sur le Build System des projets, et réalisé une analyse manuelle de certains commits.

4.1 Analyse Quantitative

L'analyse quantitative est composée de quatre dimensions, que nous allons détailler dans cette section.
Les quatre dimensions que nous avons choisies pour évaluer la qualité du Build System, avant et après la migration, sont :

1. l'évolution de la taille du Build System,
2. les modifications du code du Build System,
3. le couplage du code du Build System avec le code source,
4. l'impact des changements sur les fichiers.

Ces dimensions seront analysées à l'aide de métriques, basées sur les données récupérées dans l'intégralité de l'historique du répertoire GIT de chaque projet.

Dimensions	**Métriques**	
Evolution de la taille	BLOC	
Modifications dans le code	Taux de changement	Churn
Couplage avec le code source	Taux de co-changement	Co-churn
Impact sur les fichiers	Proportion	Distribution

4.1.1 Evolution de la taille du Build System

Cette dimension a pour objectif d'observer le développement du code du Build System. Elle permet de visualiser facilement les évenements importants de la vie du projet, comme la migration vers Maven, une période de forte expansion du code de Build System, ou une période de refactoring. C'est un premier indicateur qui donne une bonne vue d'ensemble du projet, et qui permet de cibler les mois plus intéressants à analyser qualitativement.

La métrique utilisée est le BLOC, Build Lines Of Code, qui reprend chaque mois la taille en nombre de lignes de code du Build System. Nous regardons l'évolution du BLOC de l'ancienne technologie de Build System et de la nouvelle séparément. Ce qui nous permet de trouver directement le mois de la migration, et de suivre après celle ci la façon dont le code de l'ancienne technologie continue à évoluer.

Le graphe permettant de représenter cette métrique mesure la taille en BLOC des deux Build Systems chaque mois.

Il est important de noter que l'objectif de cette métrique n'est pas simplement de comparer la taille des deux Build Systems, mais surtout la façon dont le BLOC évolue. En effet, les deux technologies peuvent avoir une verbosité différente, ce qui fait qu'une simple comparaison du BLOC des deux Build Systems n'est ni équitable, ni pertinente. Elle peut montrer une tendance, mais nous préfèrerons analyser les mouvements de la courbe du BLOC plutôt que sa hauteur.
Dans notre étude de cas, Maven est plus verbeux lorsque la configuration du Build System est personnalisée, alors qu'une configuration plus standardisée permet aux projets respectant au maximum les conventions de Maven d'avoir un BLOC plus faible.

4.1.2 Modifications du code dans le Build System

Cette dimension permet d'évaluer la quantité de travail que les développeurs doivent fournir pour maintenir le Build System. Si ce dernier doit fréquemment changer, avec des modifications de taille importante, cela signifie que la maintenance demande beaucoup de travail de la part des développeurs.
Cette dimension peut être utilisée avec la première dimension, l'évolution du code. Un pic observé dans l'évolution du BLOC peut être expliqué à l'aide de cette seconde dimension. Il pourrait être dû à de nombreux commits de petite taille, ou quelques commits de taille plus importante.

Nous utilisons deux métriques pour évaluer cette dimension, le taux de changement du code du Build System et la taille de ces changements, que nous appelons le *churn*.

La métrique du taux de changement nous permet de voir quelle était la fréquence des modifications du Build System et la proportion des commits qui le modifient le Build System.
Pour cela, nous regardons le pourcentage mensuel de commits qui ont modifiés le code du Build System. Sa valeur indique la quantité de travail de l'équipe de développement qui est dédiée à la maintenance du Build System. Plus sa valeur est élevée, plus la maintenance occupe une place importante. Il est important de conserver une fréquence de changements aussi basse que possible.

La métrique du churn est la taille des modifications qui ont été effectuées sur le Build System. Une modification peut être un ajout de ligne, une suppression de ligne ou une ligne dont le contenu a changé. Toutes ces lignes sont additionnées ensemble, et ont le même poids.

Deux graphes permettent de représenter cette métrique :
Le premier reprend pour chaque mois le nombre total de lignes modifiées dans tous les commits concernant le Build System. La taille de ces commits permet d'évaluer la quantité de travail qui a été allouée à la maintenance du Build System.
Le deuxième graphe présente la taille des commits de chaque mois sous forme de *Box Plot* (cfr. figure 4.1). Il permet d'avoir un aperçu de la taille des commits grâce aux quartiles du churn des commits de chaque mois. Ainsi le travail réalisé dans ces commits sera directement évalué. Un seul commit de plusieurs centaines de lignes demande plus de travail que plusieurs commits de quelques dizaines de lignes.

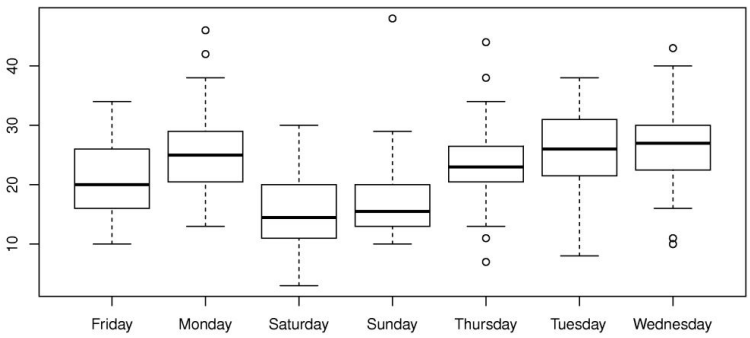

FIGURE 4.1 – Ce graphe est un exemple conceptuel de BoxPlot.
Un BoxPlot est un graphique permettant de représenter des groupes de données en utilisant leurs quartiles.

4.1.3 Couplage du code du Build System avec le code source

Cette dimension montre quelle est la corrélation entre l'évolution du code source et l'évolution du code du Build System.
Il est intéressant de savoir si la maintenance du Build System dépend directement de l'évolution du code source. Un haut couplage signifie qu'il y a de fortes chances pour que ce soient les développeurs qui modifient le Build System eux-mêmes pour que ce dernier prenne en compte leurs changements. Ceci augmente leur charge de travail et rend leur tâche plus complexe.
Par contre, un faible couplage indique que le Build System est suffisamment flexible pour s'adapter à des changements effectués dans le code source. Il réclame donc une maintenance moins systématique et moins fréquente.

Pour cette dimension, nous utilisons également comme métriques le taux de changement et le churn, mais d'un point de vue différent :
Pour le taux de co-changement, nous regardons le pourcentage de commits qui ont changé en même temps, le code source et le code du Build System. Le graphe montre, par groupe de cinquante commits modifiant le Build System, quel est le pourcentage de commits du groupe qui modifient également le code source.
Pour le co-churn, nous regardons pour chaque mois le churn cumulé des commits qui ont modifié en même temps, le code source et le code du Build System.

4.1.4 Impact des changements sur les fichiers

Cette dimension représente les fichiers du Build System qui ont nécessité une maintenance. Cela nous permet de savoir si les modifications sont réparties à travers tous les composants du Build System, ou si elles sont plus localisées, et de connaitre également la proportion des fichiers modifiés et leur nombre de modifications.
Un projet dont les changements impliquent une grande partie du Build System lors de la maintenance sera plus difficile à aborder par les développeurs. Par contre, un projet dont seuls quelques fichiers sont modifiés de nombreuses fois, et une majorité des fichiers ne nécéssitant que peu de maintenance, montre un Build System globalement plus stable. Ces maintenances sont plus localisées et plus simple à corriger pour les développeurs.

Nous utilisons deux métriques pour évaluer cette dimension, la proportion des fichiers qui ont reçu une maintenance, et la distribution de ceux-ci.

La métrique de proportion des fichiers modifiés est représentée par un graphe indiquant pour chaque trimestre le pourcentage de fichiers modifiés au moins une fois.

La métrique de distribution des modifications est représentée par une *Treemap* (cfr. figure 4.2). Cette dernière présente l'ensemble des fichiers du Build System du projet. Chaque fichier est coloré différemment selon le nombre de modifications qu'il a reçu au cours de l'année concernée.

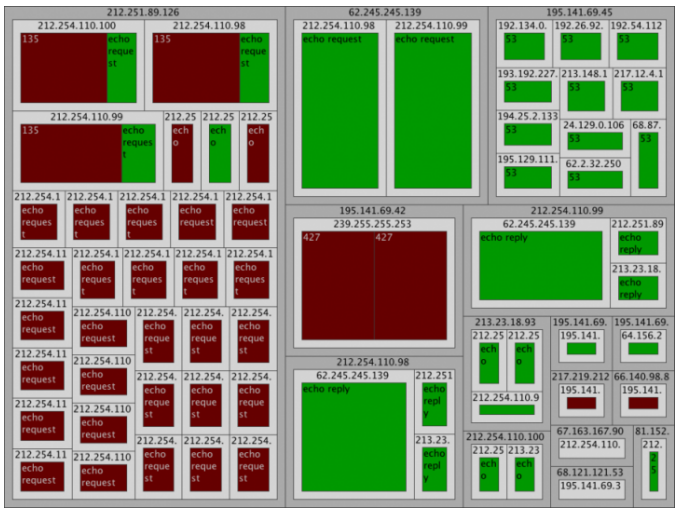

FIGURE 4.2 – Exemple conceptuel d'une Treemap.
Une Treemap est une représentation visuelle d'une arborescence de données, organisant les objets dans une structure hiérarchique.

4.2 Analyse Qualitative

La première partie de cette analyse consiste à poser aux développeurs des projets des questions pour en savoir plus sur la migration du Build System de leur projet.
L'objectif de ce sondage est de savoir quelles sont les raisons qui les ont motivé à réaliser la migration, comment ils évaluent la difficulté de cette migration et ce qu'ils pensent de Maven en listant deux points forts et deux points faibles qu'ils considèrent comme majeurs.

Voici en anglais, les questions posées :

1. Why did you choose to migrate your build system towards Maven ? Was the migration driven by the (then) current state of the existing build system, by a management decision, or by another motivation ?

2. How would you rate the difficulty to perform the migration, and why ?

3. What are the 2 majors strengths of the new Maven build system, and what are its 2 major weaknesses ?

Pour l'analyse qualitative, nous avons également effectué une analyse manuelle des commits dans l'historique de GIT. L'objectif est de mieux comprendre les événements surprenants que nous trouvons dans les graphes, principalement ceux de l'évolution du BLOC et du churn.
Les événements que nous analysons plus en détails sont par exemple, une chute du BLOC ou une brusque montée ; un grand pic de churn ne modifiant pas le BLOC.
Cette analyse manuelle permet une meilleure compréhension des raisons des changements et une évaluation plus fine des résultats.

CHAPITRE 5

Résultats

Dans ce chapitre, nous allons présenter, dans un premier temps, les réponses obtenues des développeurs, ensuite les résultats des analyses quantitatives de chaque projet et finalement l'analyse des résultats pour chaque dimension.

Il est important de noter que les Treemap de la distribution des changements ne sont pas affichées dans l'analyse individuelle des projets. En effet, les résultats de chaque projets sont très similaires pour chacun. Cette métrique sera donc uniquement illustrée et discutée dans l'analyse de sa dimension.

5.1 Réponses des développeurs

Les développeurs contactés sont ceux qui s'occupent régulièrement de la maintenance du Build System, et de préférence, ceux qui ont participé à la migration du Build System de leur projet.

5.1.1 Motivations de la migration

Les questions posées aux développeurs nous ont permis de mettre à jour les véritables raisons qui ont motivées les équipes de développement à effectuer une migration.

La raison principale évoquée est la grande complexité du Build System précédent, basé sur Ant. Par exemple, JBossTools et BPEL Designer utilisaient un outil intermédiaire qui leur permettait de réduire la complexité de Ant, mais produisait du code désordonné et peu documenté, très difficile à comprendre. Tycho quant à lui produit un code plus propre et simple à comprendre.
D'autres défauts de Ant sont sa difficulté à intégrer des unités de tests, ainsi que la difficulté d'intégration du Build System dans des serveurs d'*intégration continue* (e.g. Jenkins ou Hudson). Maven offre nativement ces fonctionnalités, au grand soulagement des développeurs.
Migrer vers Maven leur promet donc une maintenance plus aisée du Build System, grâce à des fonctionnalités plus facilement utilisable ainsi qu'un code plus propre et compréhensible.
Un Build System plus simple permet également à chaque développeur de modifier lui-même le Build System, ce qui est important dans un projet open source où l'équipe de développement n'est pas aussi stable que dans le domaine privé.

Une autre raison avancée est le manque de support actif de l'outil utilisé pour gérer le Build System. Les développeurs ont donc cherché à remplacer leur Build System, et leur choix s'est porté sur Tycho et Maven qui profitent d'un fort soutien de la part de la communauté d'Eclipse.

Une troisième raison est simplifier la vie des utilisateurs grâce à la facilité d'exécution du Build System utilisant Maven. Ces derniers peuvent ainsi télécharger le code source sur le répertoire GIT et compiler eux-même le projet sans avoir besoin de connaissance particulière du Build System.
Une simple commande *"mvn clean install"* permet de lancer l'exécution de toutes les opérations nécessaires pour construire le projet, sans besoin de configuration supplémentaire.

La dernière raison exprimée par les développeurs est que la migration vers la combinaison Tycho et Maven est encouragée par Eclipse pour ses projets, dans le but de passer à un (en anglais) "Common Build Initiative" [1], abrégé CBI.
Avant le CBI, les différents projets d'Eclipse utilisaient des Build Systems différents pour construire leur projet. L'idée de CBI est de permettre aux contributeurs de participer plus facilement à l'élaboration des différents projets d'Eclipse grâce à un Build System commun standardisé. Cette standardisation leur permet de construire et tester le projet sans devoir connaitre les particularités de son Build System.

1. http ://wiki.eclipse.org/CBI

5.1.2 Difficultés de la migration

Dans l'ensemble, les développeurs ont trouvé que la migration vers Maven en utilisant Tycho est très rapide et relativement aisée.

Il suffit connaitre, au préalable, le fonctionnement de Tycho et de Maven. Pour cela, de nombreux tutoriels et des exemples d'autres projets open source utilisant déjà Tycho et Maven existent.

La migration en elle même est très rapide, et peut être réalisée en seulement une journée par un développeur n'ayant pas encore d'expérience avec Tycho. Une fois le processus de migration maitrisé, elle peut même être réalisée encore plus rapidement.
Tycho permet de générer automatiquement plusieurs parties qu'il fallait autrefois coder manuellement.
De plus, la communauté de Tycho est très active, et de nombreuses fonctionnalités sont développées pour faciliter la migration. Par exemple, lorsque JBoss Tools a débuté sa migration, c'était l'un des premiers à utiliser Tycho et il n'était pas possible de générer automatiquement les fichiers pom.xml.
À présent, Tycho dispose d'une commande simple permettant de générer une version simple de tous les fichiers POM nécessaires :
"mvn org.eclipse.tycho :tycho-pomgenerator-plugin :generate-poms -DgroupId=<MY.GROUPID>"[2]. Ces fichiers seront bien entendu à peaufiner manuellement après la génération pour les adapter au projet.

L'opération la plus longue est celle qui consiste à ajuster la structure du projet à la structure standardisée par Maven : un dossier pour les plug-ins, un pour les fonctionnalités, un pour les tests, etc.

2. http ://wiki.eclipse.org/Tycho/Reference_Card#Generating_POM_files

5.1.3 Forces et faiblesses de Maven

Les principales forces de Maven citées par les développeurs sont :
- Le téléchargement automatique des plug-ins et des dépendances, à l'aide des répertoires centralisées comme Maven Central Repository.
- La gestion explicite des targets et des dépendances qui rendent les builds raisonnablement aisés à reproduire.
- La nature déclarative des fichiers POM qui facilitent la configuration.
- Le plug-in permettant à Maven d'utiliser du code Ant pour conserver des parties du précédent Build System Ant, le temps de trouver comment les réaliser avec Maven ou attendre que ces fonctionnalités apparaissent dans la technologie.
- La simplicité de l'exécution du build en local et avec les systèmes d'intégration continue.
- La commande *"mvn clean install"* commune à tous les projets, qui permet de construire n'importe quel projet.

Les principales faiblesses de Maven citées par les développeurs sont
- Les phases du cycle de vie optionnelles ne sont pas faciles à comprendre et à contrôler. Lorsqu'une étape est ajoutée, il n'est pas évident de savoir quand exactement elle sera réalisée. Il est également plus difficile de simplement exécuter un aspect particulier du Build System, par exemple télécharger un seul fichier *jar* sur lequel le projet dépend.
- L'exécution du build est lente.
- Il y a des duplications dans les fichiers POM et dans les Manifests.
- Lorsqu'une dépendance ne se trouve pas dans les répertoires centralisés, il est n'assez aisé de l'inclure dans le Build System.
- Maven a encore quelques limitations, qui obligent l'utilisation du code Ant pour certaines fonctionnalités. Dans ce cas, le code exécuté par le plug-in utilisant le code Ant n'est ni vu, ni contrôlé par Maven.
- La documentation n'est pas toujours disponible concernant la syntaxe à utiliser. Heureusement, il y a des channels IRC [3], des newsgroups et Google pour trouver les renseignements nécessaires.

3. Internet Relay Chat - sert à la communication instantanée principalement sous la forme de discussions en groupe par l'intermédiaire de canaux de discussion

5.2 Présentation des résultats

5.2.1 LinuxTools

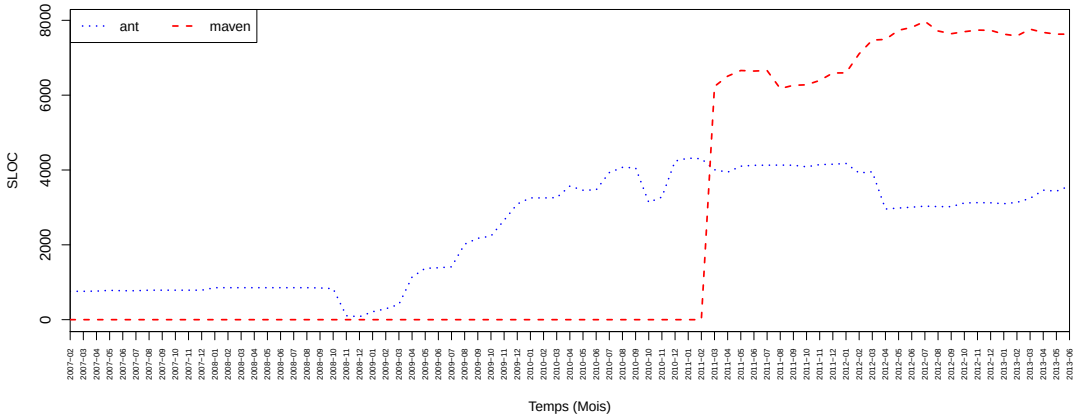

FIGURE 5.1 – BLOC de LinuxTools

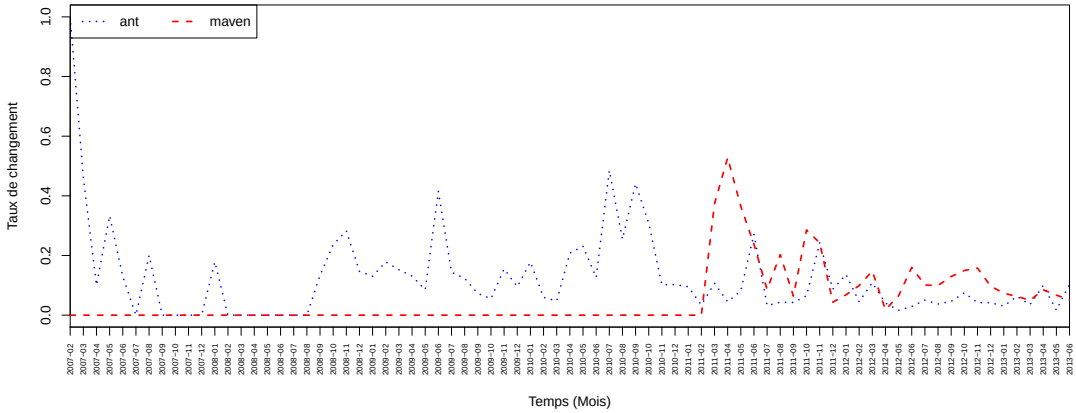

FIGURE 5.2 – Taux de changement de LinuxTools

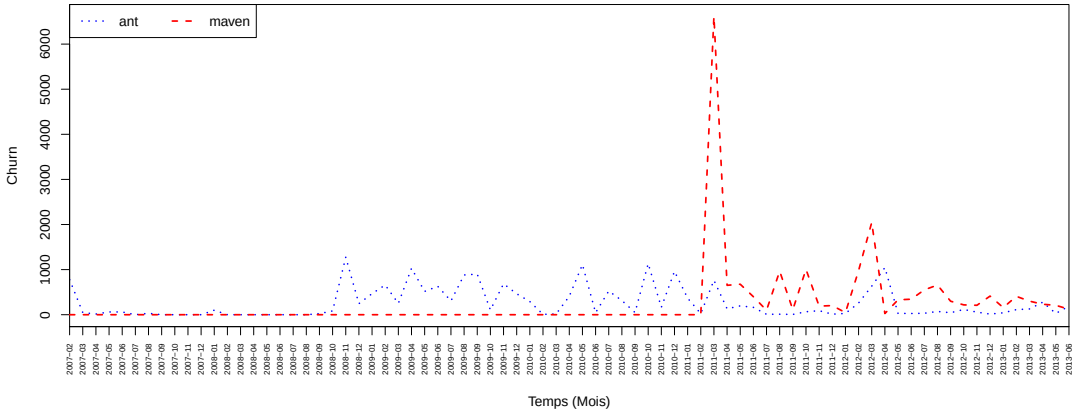

FIGURE 5.3 – Churn de LinuxTools

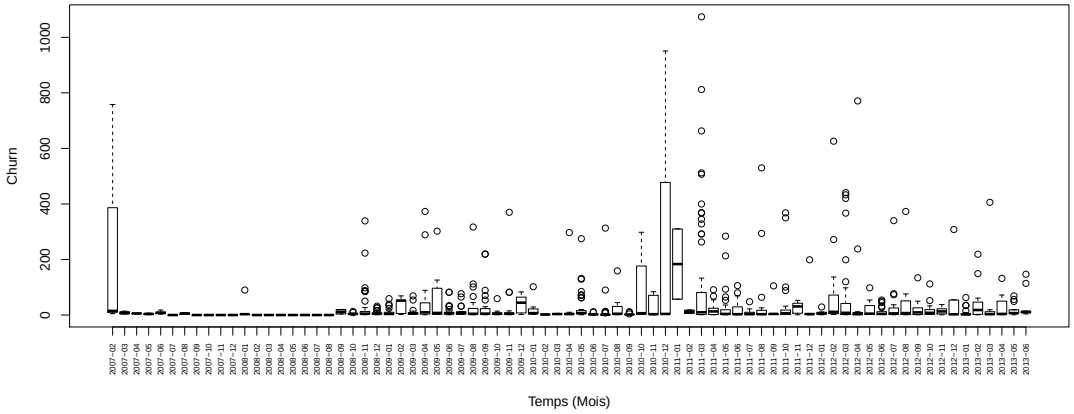

FIGURE 5.4 – Churn de LinuxTools sous forme de Boxplot

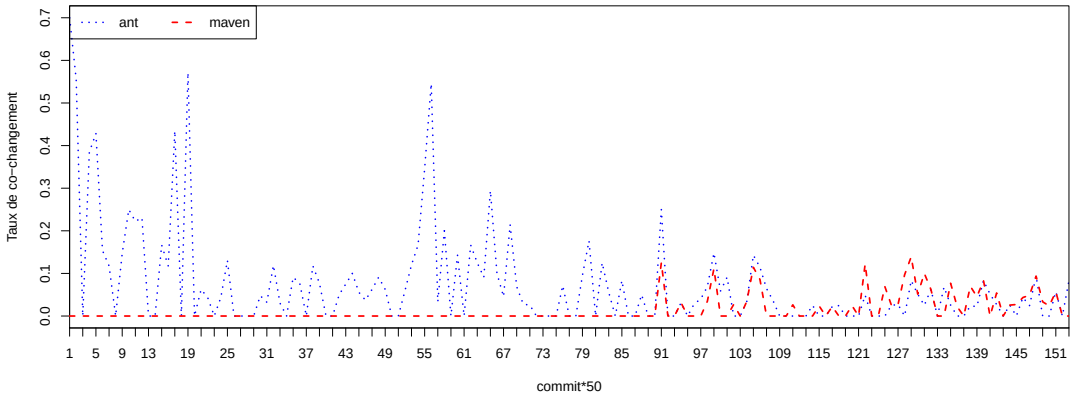

FIGURE 5.5 – Taux de co-changement de LinuxTools

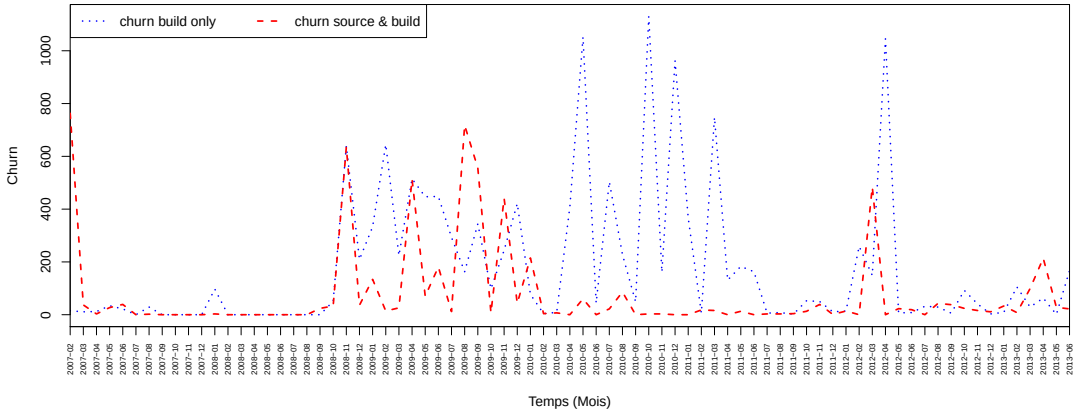

FIGURE 5.6 – Co-churn de Ant de LinuxTools

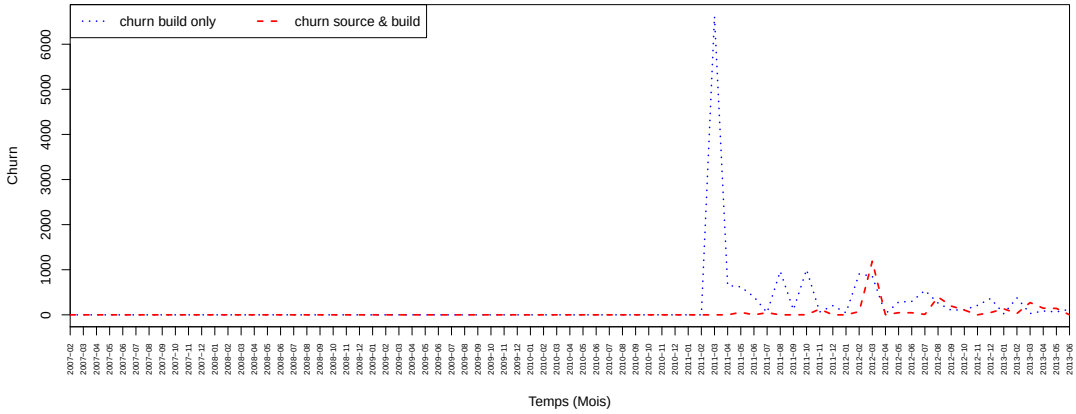

FIGURE 5.7 – Co-churn de Maven de LinuxTools

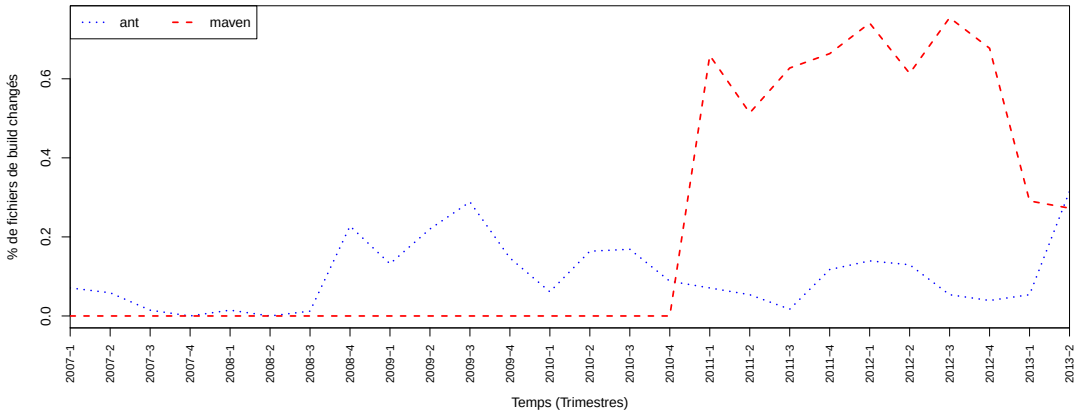

FIGURE 5.8 – Impact sur les fichiers de Build System de LinuxTools

Le Build System Maven grandit les 18 premiers mois, puis reste stable la dernière année.
La figure 5.1 nous montre l'évolution du BLOC de LinuxTools.
Le Build System Ant commence par une période stable, avec un nombre de lignes de code qui évolue de 750 à 800 lignes pendant les deux premières années. Il y a très peu d'historique pour cette période, la majorité s'est perdue lors du changement vers le répertoire GIT actuel. De ce fait, il y a peu de changement présent dans les graphes pour cette période.
Ensuite, son évolution prend de l'ampleur et le nombre de lignes de code évolue de façon constante au fil des mois, jusqu'à dépasser les 4000 lignes avant la migration. La chute d'un peu plus de 900 lignes en octobre 2010 est due à une restructuration, alors que la forte remontée d'environ 1000 lignes en décembre 2010 est due à l'ajout de trois fichiers build.xml dans les dossiers de tests.

La migration vers Maven se déroule en mars 2011 avec un *big bang*, l'ensemble du code a été écrit pendant le mois de la migration. La taille atteint alors les 6000 lignes de code, soit une augmentation de 50% par rapport au code Ant. La taille du code Maven reste stable pendant quelques mois avant de connaitre une première chute de 400 lignes en août 2011 due à une restructuration du Build System, où les développeurs ont retiré beaucoup de code n'étant plus nécessaire et causant des avertissements. Ensuite la taille monte progressivement pendant un an, culminant près des 8000 lignes de code, avant de se stabiliser pendant l'année qui suit, jusqu'en juin 2013.

Après la migration, le code Ant est toujours présent et régulièrement mis à jour, car il est utilisé par le code Maven pour des fonctionnalités encore indisponibles avec Maven.
La taille du code Ant n'évolue pas beaucoup, à l'exception d'une chute de 1000 lignes de code en avril 2012, soit 25% de sa taille. Celle-ci est causée par le retrait de code obsolète permettant de lancer les tests, cette fonction étant à présent prise en charge par le code Maven. Ensuite, le code Ant connait une légère croissance.

La première année, le code Maven a un taux de changement similaire à celui de Ant avant la migration, mais ensuite il est bien plus bas.
La figure 5.2 nous montre le taux de changement du Build System de LinuxTools.
Jusqu'en septembre 2008, le peu d'historique disponible ne permet pas d'avoir des résultats pertinents.
Le taux mensuel de changements de Ant avant la migration est majoritairement dans les 18%, et quelques pics au dessus de 30%, et même jusqu'à 48% en juillet 2010. Le taux moyen entre septembre 2008 et début 2011 est de 17%.

Lors de la migration vers Maven, les trois premiers mois ont requis beaucoup de travail, une migration d'un Build System de cette taille n'étant pas triviale. Il est donc normal de voir des taux de 37%, 52% et 38% sur ces mois. Le code connait encore du changement durant les six mois suivants, avec certains mois ayant des taux entre 20% et 28%. A partir de décembre 2011, le Build System se stabilise et la maintenance ne demande plus autant de travail. La majorité des taux mensuels sont inférieurs à 10%, et les plus élevés sont inférieurs à 16%.

La dernière année de développement connait d'ailleurs un taux de changement particulièrement bas, avec un taux moyen de seulement 5.8%.

Après la migration, le code Ant est toujours régulièrement mis à jour, mais mis à part deux pics à 27% et 24%, le taux est majoritairement inférieur à 5%.

Nous constatons donc les premiers effets bénéfiques de la migration. Après une première année post-migration de modification assez élevée, la situation se stabilise autour d'un taux de changement bien inférieur à celui du Build System Ant avant la migration.

Le churn de Maven est similaire à celui de Ant la première année, ensuite il devient significativement inférieur.
La figure 5.3 nous montre le churn du Build System de LinuxTools.
A partir de septembre 2008, la taille des changements est régulièrement élevée, avec des mois voyant un churn entre 500 et 1200 lignes cumulées. Mais certains mois montrent un churn très faible, malgré parfois un taux de changement assez élevé, comme en août 2010 où le churn est de 300 lignes alors que le taux est de 25%.

La migration initiale se fait en un bloc de 6600 lignes de code Maven. Les mois qui suivent la migration ont un churn de 600 lignes, alors que le taux de changement était très élevé, 52% et 38%. Les nombreux changements ont donc été concentrés sur du debug du Build System. Le pic de 2000 lignes en mars 2012 est dû à l'intégration du projet LTTng2 dans le projet. La dernière année connait un churn relativement bas, constamment inférieur à 500 lignes.

La figure 5.4 nous présente le churn du Build System de LinuxTools par une évolution de Box Plot.
Nous pouvons constater que les quantiles sont généralement très bas. En effet, près de 80% des commits ont une taille inférieure à 20 lignes modifiées. Dans les commits de plus grande taille, avant la migration le code Ant n'a connu que 12 commits dont le churn dépassait 200 lignes, avec un maximum de 400 lignes.
La migration s'est effectuée avec de nombreux commits de grande taille, entre 300 et 1100 lignes.
Après la migration, le Build System connait plus d'une vingtaine de commits supérieurs à 200 lignes, avec un maximum de 800 lignes.

Le churn de Maven est donc relativement similaire à celui de Ant lors de la première année qui a suivi la migration, mais une fois la maintenance stabilisée lors de la dernière année, il est significativement inférieur à Ant. Les changements de Maven se font par contre plus régulièrement avec des commits de plus grande taille que Ant.

Le code Maven semble beaucoup moins dépendant du code source que ne l'est le code Ant.
La figure 5.5 nous montre le couplage entre le code source et le code du Build System de LinuxTools.
Le code Ant a régulièrement des commits couplés au code source, dépassant les 40% des commits à trois reprises et atteignant régulièrement les 20%. De son coté, Maven reste beaucoup plus bas, atteignant au maximum les 15%.

Le code Ant connait des pics assez élevés de co-churn, alors que celui de Maven est beaucoup plus bas.
La figure 5.6 et la figure 5.7 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Nous remarquons que de novembre 2008 à février 2010, le churn des commits couplés de Ant est aussi élevé en général que celui qui ne modifie que le Build System. Les pics montent jusqu'à 800 lignes couplées. Ensuite jusqu'en août 2012, le churn des commits couplés est très bas, mis à part l'import du projet en mars 2012 et un pic de 200 lignes couplées en avril 2013.
Le code Maven quant à lui connait peu de churn couplé la première année, mais celui ci monte à partir d'août 2012, et est semblable au churn indépendant du code source. Le pic de 1200 lignes en mars 2012 est lui aussi dû à l'import du projet, ce n'est donc pas de la maintenance. En dehors de ce pic, le churn couplé ne dépasse pas les 400 lignes.

Les fichiers de Maven sont beaucoup plus impactés par les changements.
La figure 5.8 nous montre l'impact des changements sur les fichiers du Build System de LinuxTools.
Nous pouvons constater que les modifications de Ant ne modifient que 30% des fichiers Ant du Build System au maximum, avec des périodes plus creuses. Par contre, Maven impact constamment entre 50 et 70%.
Seuls les deux derniers trimestres voient un impact plus réduit, de 30%.

En conclusion, la migration semble avoir pris du temps avant de se stabiliser, mais la dernière année nous montre un Build System demandant significativement moins de modifications du Build System, et des modifications d'une taille cumulée bien plus petite. Le Build System a également gagné en indépendance par rapport au code source, principalement ur le taux de changement. Le co-churn n'a proportionnellement pas beaucoup diminué, comme le montre la dernière année sur la figure 5.7. Maven montre également des modifications réparties sur une part beaucoup plus importantes des fichiers du Build System.

5.2.2 CDT

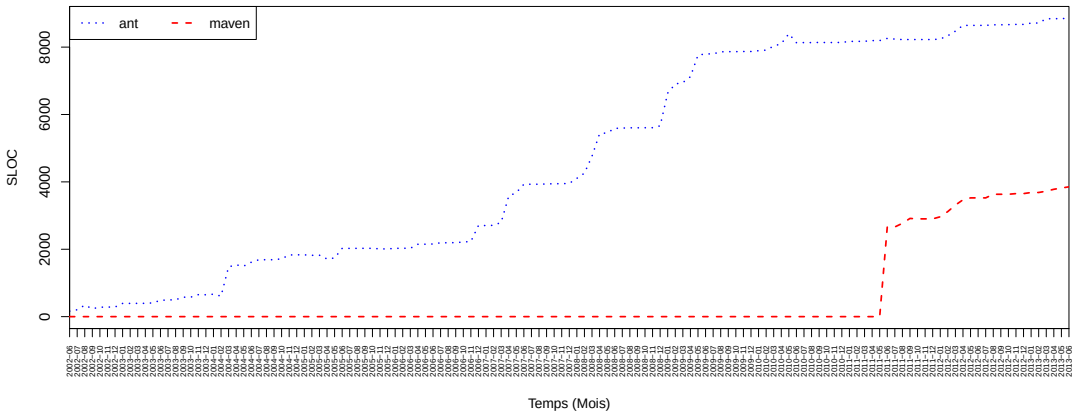

FIGURE 5.9 – BLOC de CDT

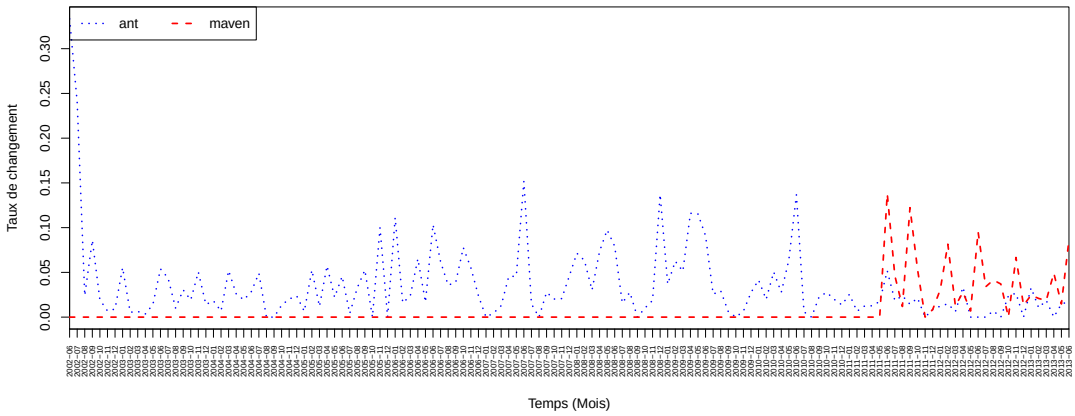

FIGURE 5.10 – Taux de changement de CDT

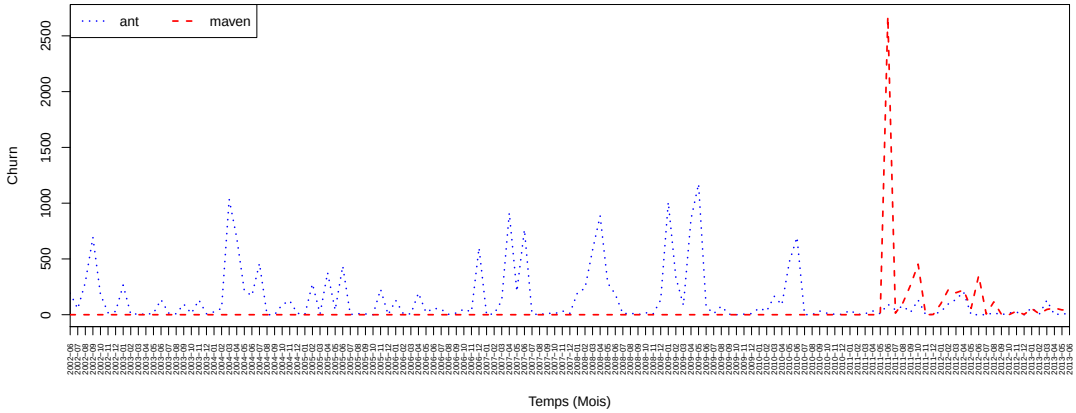

FIGURE 5.11 – Churn de CDT

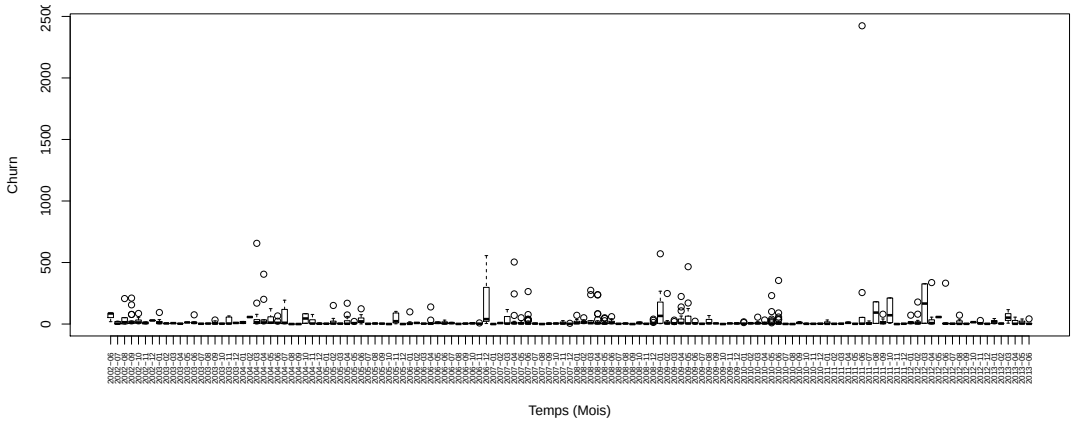

FIGURE 5.12 – Churn de CDT sous forme de Boxplot

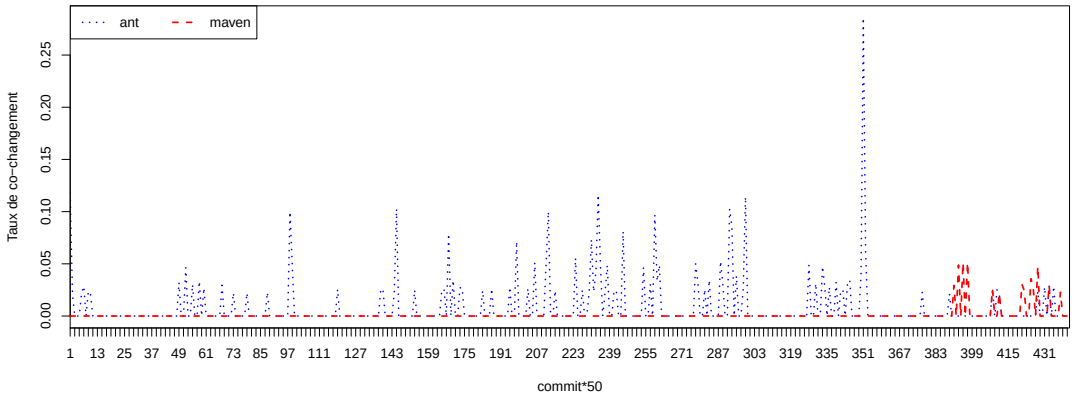

FIGURE 5.13 – Taux de co-changement de CDT

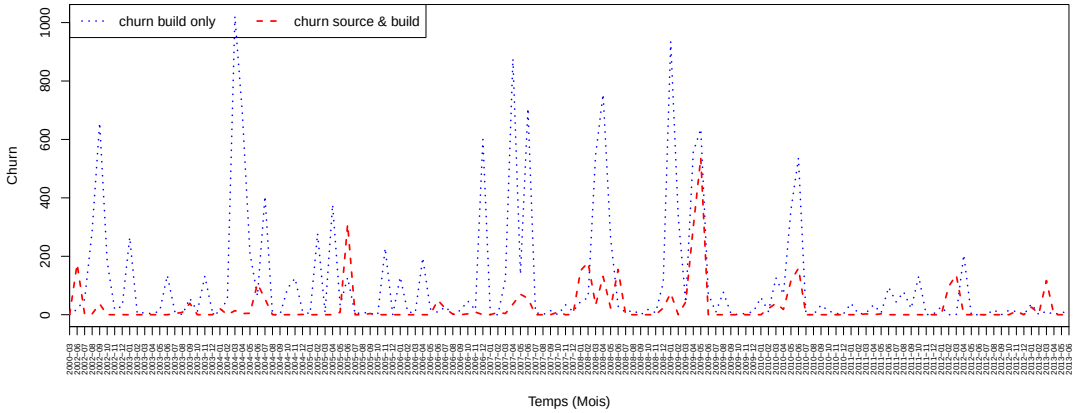

FIGURE 5.14 – Co-churn de Ant de CDT

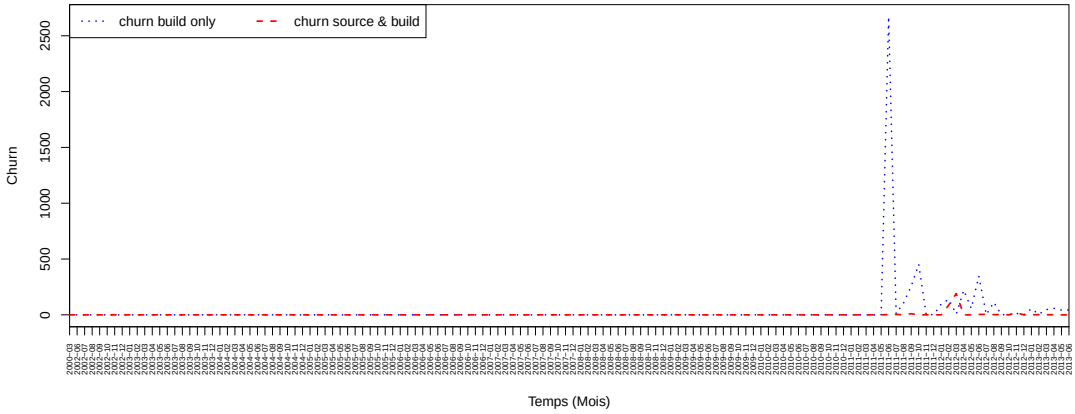

FIGURE 5.15 – Co-churn de Maven de CDT

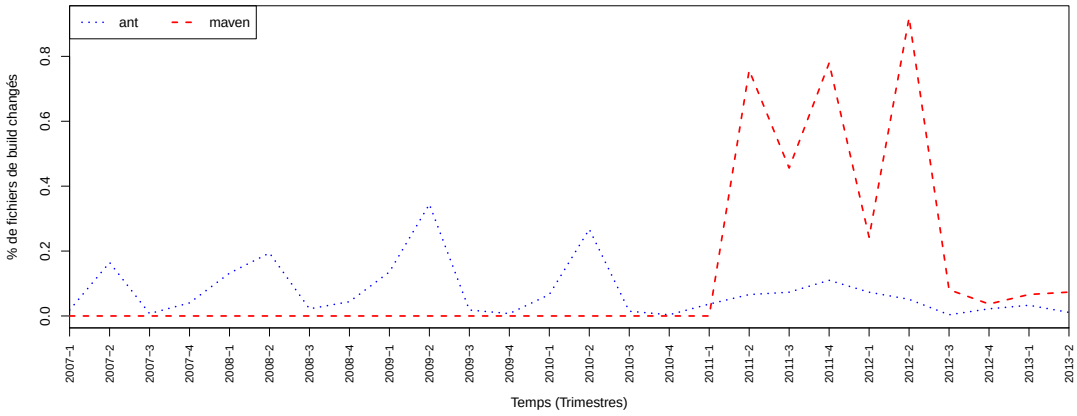

FIGURE 5.16 – Impact sur les fichiers de Build System de CDT

Le Build System Maven est beaucoup plus petit, et a une faible croissance.
La figure 5.9 nous montre l'évolution du BLOC de CDT.
Le Build System Ant croît de façon constante jusqu'en mai 2009, avec une croissance d'abord assez faible jusqu'en décembre 2006, en dépassant les 2000 lignes en quatre ans et demi. De janvier 2006 jusqu'à mai 2009, période pendant laquelle la croissance de Ant est la plus forte, le Build System croît de 2000 lignes jusqu'à 8000 lignes. Ensuite, la croissance se stabilise et ne monte plus que très faiblement, ne montant que de 500 lignes au total jusqu'en juin 2013.
La migration a eu lieu en juin 2011, avec un *big bang* de 2500 lignes. Ensuite, le Build System Maven croît peu, en prenant 1000 lignes sur deux ans, jusqu'en juin 2013. Le Build System Maven fait donc 45% de la taille du Build System Ant en juin 2013.

Le taux de changement de Maven est similaire à celui de Ant, et décroit légèrement.
La figure 5.10 nous montre le taux de changement du Build System de CDT.
Le Build System Ant change peu jusqu'en octobre 2005, avec un taux de changement constamment entre 1 et 5%. Ensuite jusqu'en juillet 2010, les changements sont plus fréquents, ceux ci montant jusqu'à 15%, et sont régulièrement au dessus des 5%. Par contre, nous observons des changements par pics, avec des périodes où le Build System ne change presque pas, avec un taux entre 0 et 3%. Après juillet 2010, comme le montrait l'évolution du BLOC, le Build System Ant ne change plus beaucoup, il ne dépasse plus les 5%.
Le Build System Maven évolue également avec des pics, qui sont de plus en plus bas. Le premier pic en juin 2011, à la migration, est à 14% alors que les derniers sont en dessous de 10%.
La situation ne s'est donc pas vraiment améliorée du point de vue du taux de changement, mais celui ci était déjà bas. La situation reste donc très bonne.

Le churn de Maven est largement inférieur à celui de Ant.
La figure 5.11 nous montre le churn du Build System de CDT.
Le Build System Ant évolue encore une fois par pics, comme le montrait déjà le taux de changement, mais ceux ci sont plus espacés que dans le taux de changement. Les pics sont entre 500 et 1200 lignes modifiées.
Le Build System Maven, une fois la migration passée, n'évolue pas beaucoup. Il y a seulement trois pics, un de 300 lignes en octobre 2010, un de 200 lignes de février à avril 2012, et un de 250 lignes en juin 2012. Le reste du temps, les changements sont de taille minime.

La figure 5.12 nous présente le churn du Build System de CDT par une évolution de Box Plot.
La majorité des commits sont de petite taille. Il y a quelques commits entre 200 et 700 lignes répartis dans l'ensemble du développement. La migration vers Maven s'est faite d'un seul coup, avec un commit d'un peu moins de 2500 lignes. Les pics de churn de Maven ont souvent été faits en peu de commits.

Maven est plus indépendant du code source que Ant, bien qu'il le soit déjà fortement.
La figure 5.13 nous montre le couplage entre le code source et le code du Build System de CDT. Le co-changement est relativement faible, très souvent autour de 3% des commits sont couplés à des changements de code source. Il y a malgré tout une dizaine de pics montant à 10%, et un pic de 28%. Maven est fortement indépendant du code source, seulement 5% au maximum est couplé.

Maven a très peu de co-churn, alors que Ant en a régulièrement.
La figure 5.14 et la figure 5.15 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Les commits couplés au code source ayant un churn élevé ne sont pas nombreux, mais sont en pics de 100 à 200 lignes de code. Il y a un pic de 300 lignes en juin 2005 et un pic de 550 lignes en mai 2009.
Maven quant à lui n'à qu'un seul pic de février (87 lignes) à mars 2013 (191 lignes).

Les fichiers de Maven sont beaucoup plus impactés par les changements, mais se stabilise à 15% la dernière année.
La figure 5.16 nous montre l'impact des changements sur les fichiers du Build System de CDT. Les modifications du Build System Ant impacte les fichiers Ant à hauteur de 40% des fichiers au maximum. Les changements se font par vagues, le deuxième trimestre de chaque année ayant le plus haut pic.
Du premier trimestre de 2011 jusqu'au troisième trimestre de 2012, le Build System de Maven a beaucoup été impacté par les changements, avec un minimum de 30% des fichiers touchés et un maximum de 80%. Ensuite, les maintenances deviennent plus localisées et ne touchent plus qu'un peu moins de 15% des fichiers sur la dernière année.

En conclusion, la migration de CDT semble être un succès. La taille du Build System est cette fois ci plus basse que celle de Ant, ainsi que le churn de Maven qui est globalement plus bas que celui de Ant. La situation reste semblable pour le taux de changement et le couplage avec le code source, mais elle était déjà très positive avant la migration.

5.2.3 BPEL Designer

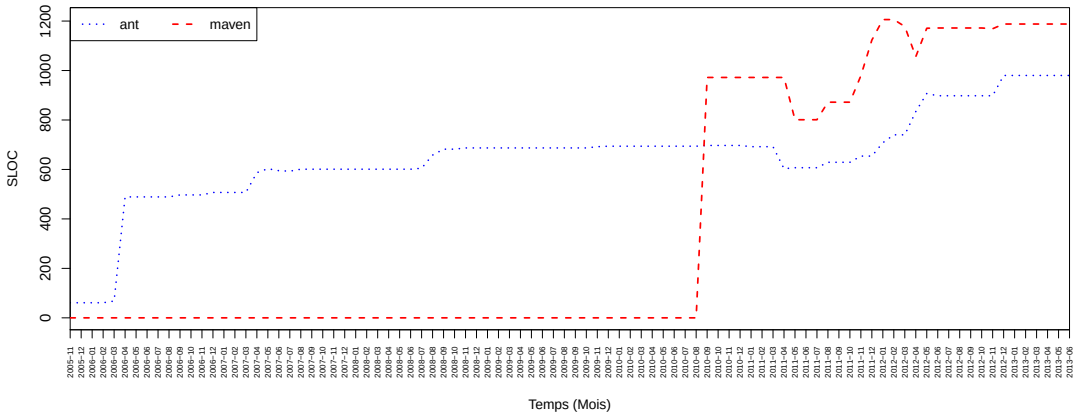

FIGURE 5.17 – BLOC de BPEL Designer

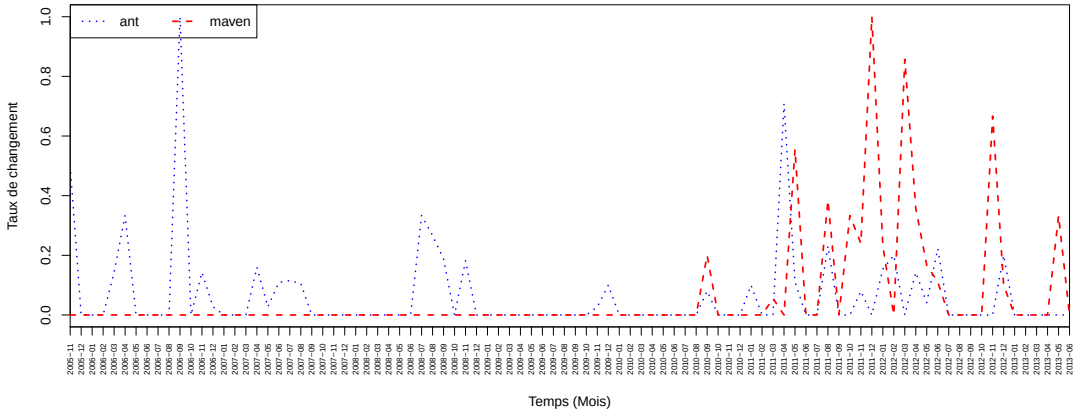

FIGURE 5.18 – Taux de changement de BPEL Designer

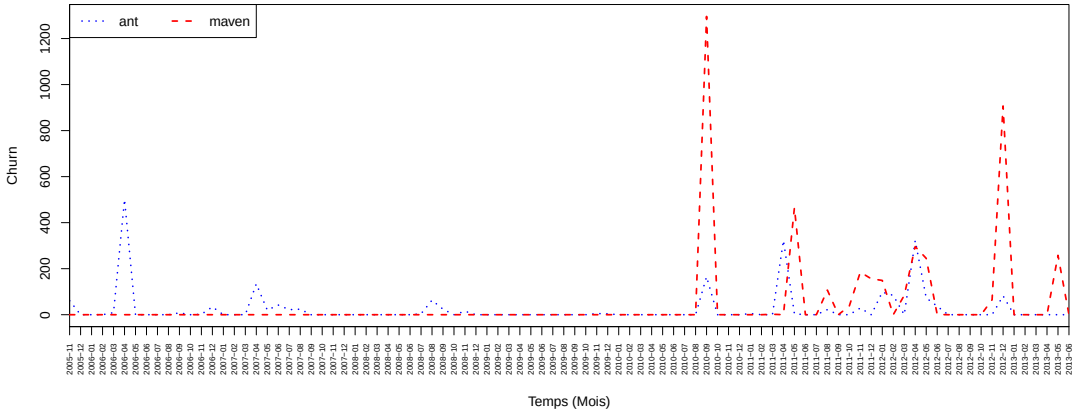

FIGURE 5.19 – Churn de BPEL Designer

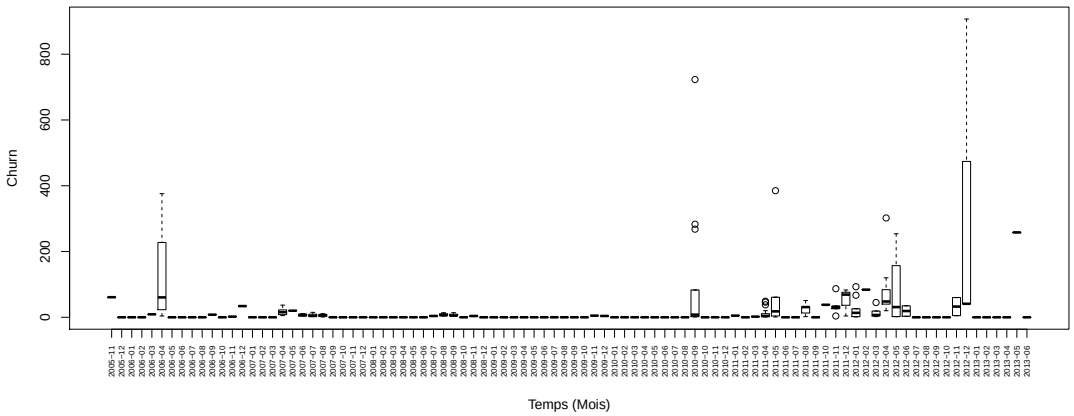

FIGURE 5.20 – Churn de BPEL Designer sous forme de Boxplot

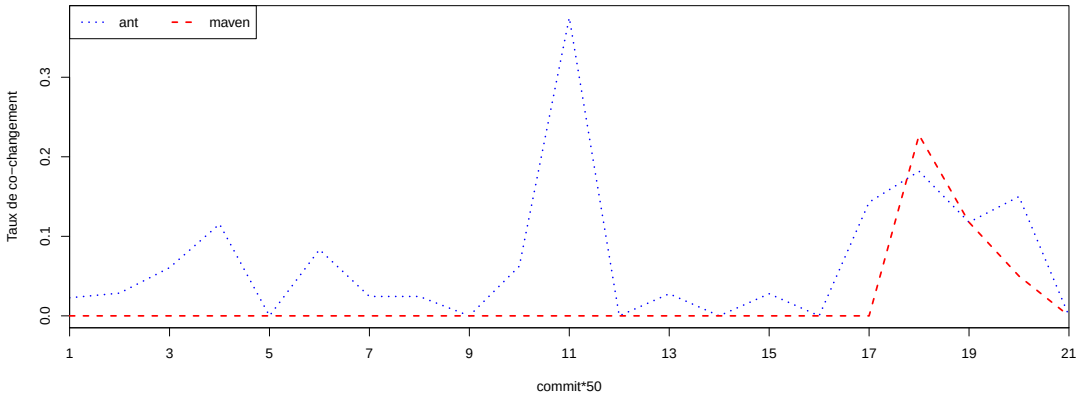

FIGURE 5.21 – Taux de co-changement de BPEL Designer

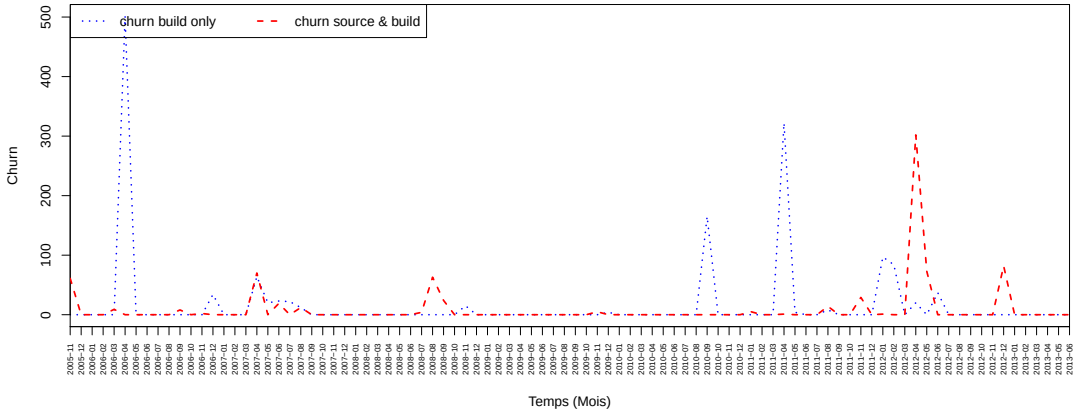

FIGURE 5.22 – Co-churn de Ant de BPEL Designer

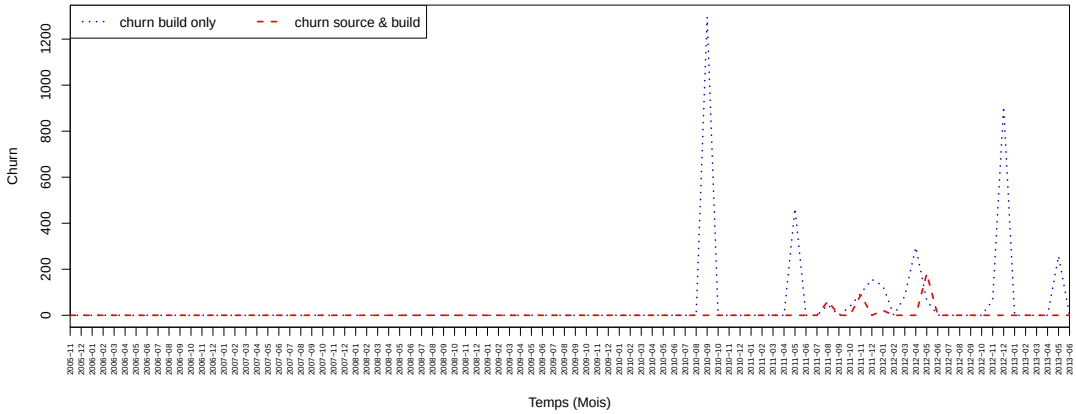

FIGURE 5.23 – Co-churn de Maven de BPEL Designer

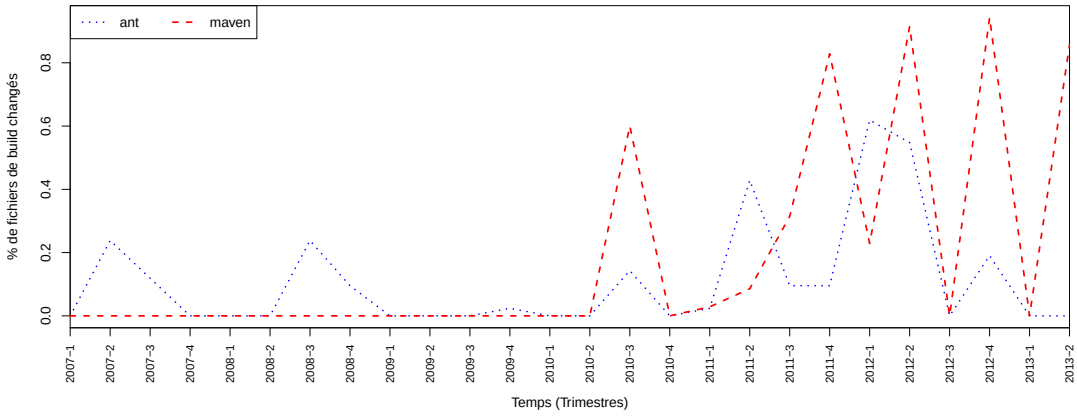

FIGURE 5.24 – Impact sur les fichiers de Build System de BPEL Designer

La migration entraine plus de changements dans le BLOC.
La figure 5.17 nous montre l'évolution du BLOC de BPEL.
L'évolution du Build System Ant de BPEL est très calme, avec trois croissances principales du BLOC, une première de 400 lignes en avril 2006, et deux de 100 lignes en avril 2007 et août 2008.
La migration a eu lieu en septembre 2010, avec un *big bang* de 900 lignes de code. Ensuite, aussi bien le Build System Ant que Maven subissent plusieurs modifications, dont deux chutes de BLOC pour Maven et une pour Ant.

Le taux de changement de Maven est bien plus élevé que celui de Ant.
La figure 5.18 nous montre le taux de changement du Build System de BPEL.
Comme le laissait présager l'évolution du BLOC, la période pré-migration connait de nombreuses périodes calmes. Par contre, lorsque le Build System est modifié, cela semble prendre parfois une grosse partie de l'effort du mois des développeurs, comme en septembre 2006 avec 100% des commits modifiant le Build System, et environ 35% en avril 2006 et juillet 2008.
Après la migration, la maintenance du Build System, aussi bien la partie Maven que Ant, prend une part bien plus importante. Les modifications sont beaucoup plus fréquentes et nécessitent beaucoup d'efforts. Quatre mois ont un pourcentage supérieur à 55%, allant jusqu'à 100% à nouveau en décembre 2011.

Le churn de Maven est bien plus élevé que celui de Ant.
La figure 5.19 nous montre le churn du Build System de BPEL.
La période pré-migration est très calme. Même en septembre 2006 où le taux de changement était de 100%, les modifications apportées étaient en réalité négligeable en taille. Le code Maven nécéssite de grande modifications de temps en temps et le reste du temps la situation est assez calme. En effet, la plupart des gros changements sont répartis sur six pics, et le Build System change peu en dehors des pics.
Il y a un pic de 900 lignes en décembre 2012 qui passe inaperçu dans l'évolution du BLOC. C'est dû à une modification d'un fichier pom.xml, dont toutes les lignes ont été supprimées puis réécrites alors que seule une petite partie devait réellement changer.

La figure 5.20 nous présente le churn du Build System de BPEL par une évolution de Box Plot.
Les commits des mois avec des pics sont souvent de grande taille, avec plusieurs quartiles entre 100 et 200 lignes, et certains commits allant de 400 à 900 lignes.

Pour la période post-migration, le taux de co-changement de Ant et Maven sont similaires.
La figure 5.21 nous montre le couplage entre le code source et le code du Build System de BPEL.
Le code Ant est relativement bien indépendant du code source pendant la période pré-migration, les modifications communes étant la majorité du temps en dessous des 10%, excepté un pic à 35%. A partir des cinquante commits avant la migration et toute la période post-migration, Ant devient plus dépendant, étant à chaque fois entre 15 et 20%.
Maven quant à lui a eu un pic à 20% lors de la migration, mais la dépendance avec le code source décroit rapidement dès le groupe de cinquante commits suivant, qui est à 10%.

Le co-churn de Maven est plus faible que celui de Ant.
La figure 5.22 et la figure 5.23 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Le co-churn de Ant est très faible pendant la majorité du développement, excepté en avril 2012 où il y a un pic de 300 lignes dans des commits changeant également le code source. La plupart des changements couplés au code source détectés dans le co-changement n'étaient donc que des petites modifications.
La situation est similaire pour Maven, il n'y a que trois pics de 100 à 200 lignes, le reste du temps les lignes modifiées de façon couplée sont presque inexistantes.

Les fichiers de Maven sont beaucoup plus impactés par les changements.
La figure 5.24 nous montre l'impact des changements sur les fichiers du Build System de BPEL.
Les fichiers d'Ant ne sont pas globalement souvent modifiés, seuls 30% au maximum des fichiers sont modifiés jusqu'au premier trimestre de 2012. Un pic à 60% survient le trimestre suivant, puis le Build System Ant n'est plus modifié.
Pour Maven, les modifications se font par pics, ce qui confirme les observations des précédentes métriques. Les pics par contre montent assez haut, de 60 à 80% des fichiers sont modifiés à chaque pic.

En conclusion, la charge de travail n'a pas été diminuée suite à la migration, mais cela semble être l'inverse.
Cette situation n'est pas entièrement à imputer à la migration, car l'analyse des commits montre qu'une collaboration avec JBoss a démarré fin 2010, soit dans les environs de la migration. Cela implique plus d'effort à effectuer sur le Build System pour que BPEL Designer puisse être facilement intégré au projet.
Le feedback du développeur en charge de la migration nous a révélé que le Build System Ant était réellement difficile à modifier, alors que le nouveau Build System Maven est beaucoup plus abordable. La migration leur a donc permis d'avoir un Build System plus simple à modifier pour étendre BPEL à d'autres projets, et les développeurs en ont donc profité.

5.2.4 JDT-Core

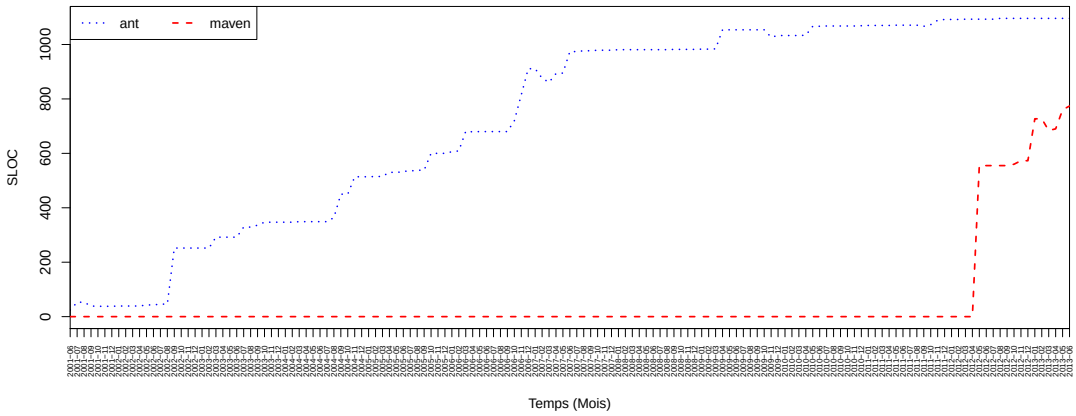

FIGURE 5.25 – BLOC de JDT-Core

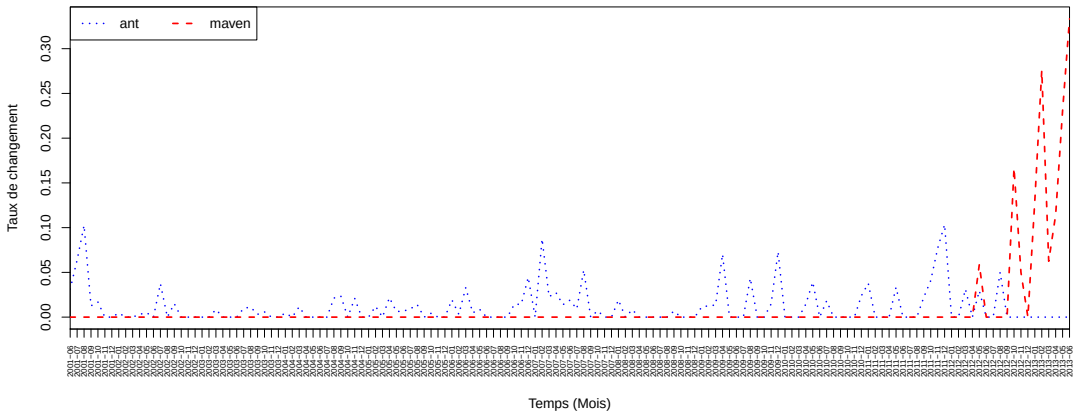

FIGURE 5.26 – Taux de changement de JDT-Core

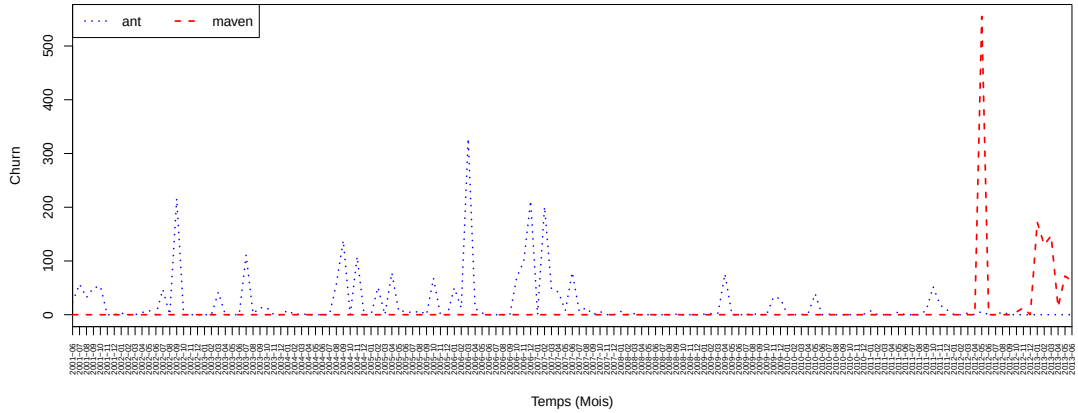

FIGURE 5.27 – Churn de JDT-Core

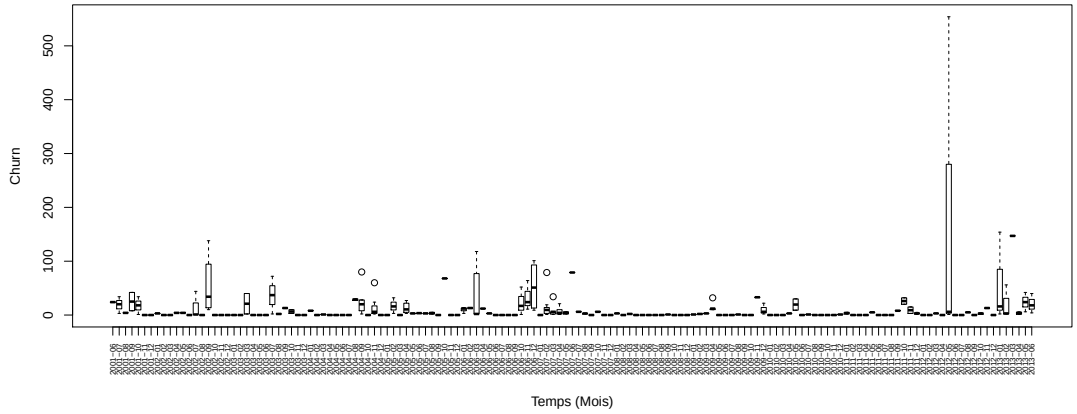

FIGURE 5.28 – Churn de JDT-Core sous forme de Boxplot

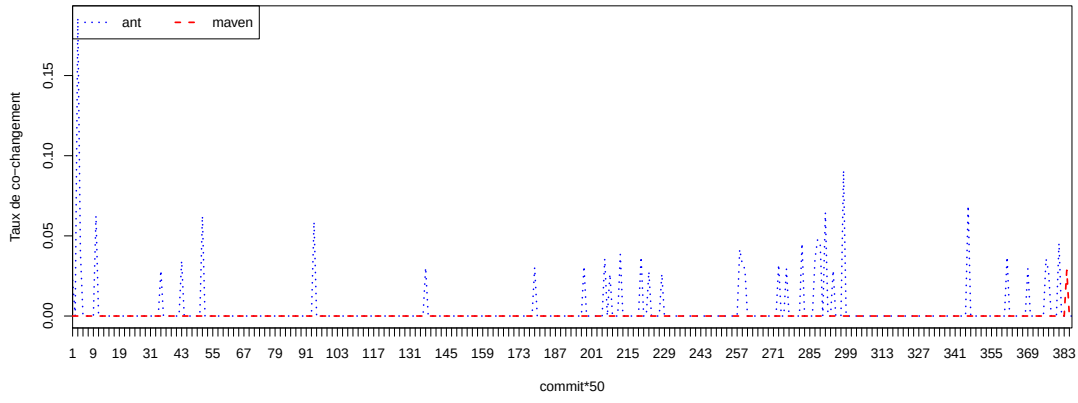

FIGURE 5.29 – Taux de co-changement de JDT-Core

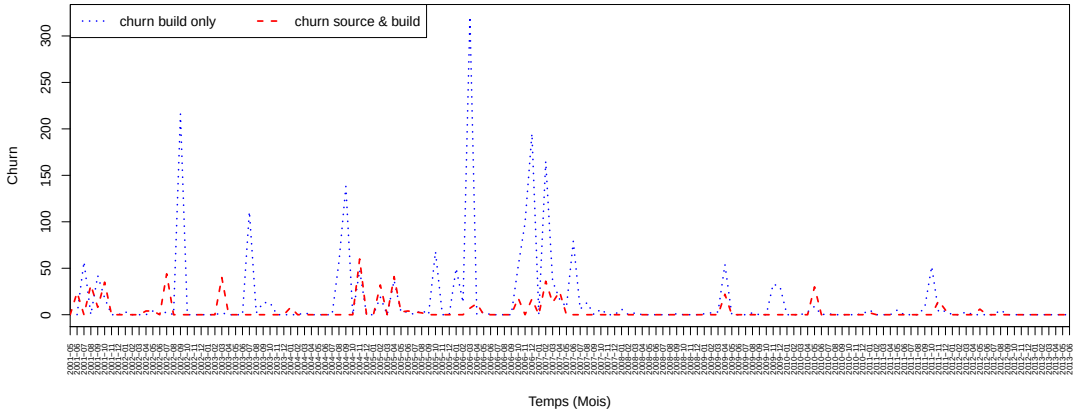

FIGURE 5.30 – Co-churn de Ant de JDT-Core

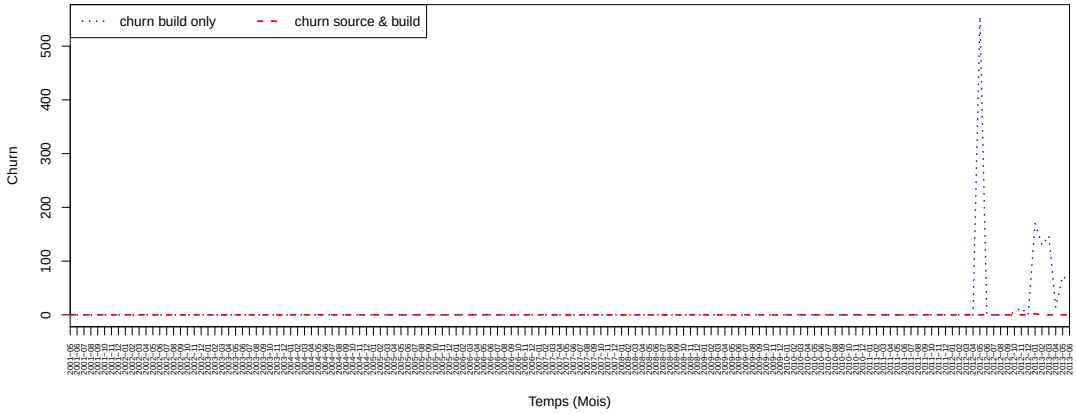

FIGURE 5.31 – Co-churn de Maven de JDT-Core

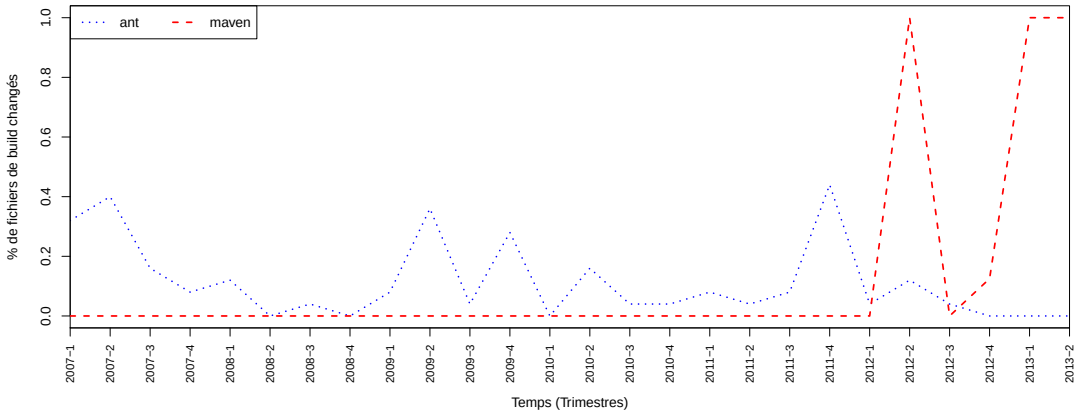

FIGURE 5.32 – Impact sur les fichiers de Build System de JDT-Core

Le Build System Maven est largement plus petit, mais a une croissance assez brusque.
La figure 5.25 nous montre l'évolution du BLOC de JDT-Core.
Le Build System Ant croît de façon constante jusqu'en juin 2007, puis se stabilise autour d'un millier de lignes de code. La croissance est en plateau, avec de grandes croissances sur un ou deux mois, puis quelques mois de faible croissance.
La migration a lieu en mai 2012 par un big bang de 500 lignes de codes, et le Build System Maven croît proportionnellement assez vite, de 200 lignes de code sur les mois suivants. Le code Maven est inférieur de 30% en taille par rapport à celui de Ant.

Le taux de changement de Maven est plus élevé, et ne cesse d'augmenter au fil des mois.
La figure 5.26 nous montre le taux de changement du Build System de JDT-Core.
Le Build System Ant n'est pas modifié tous les mois, il y a de nombreux espaces de quelques mois pendant lesquels le Build System n'est presque pas modifié. Les mois avec les plus forts pourcentages de modifications sont de 5 à 10% des commits du mois.
Le Build System Maven quant à lui change presque tous les mois, et prend de plus en plus d'effort. La migration prend 7% des commits, puis un pic à 15% en novembre 2012, suivi d'un pic à 25% en février 2013 et un pic à 30% en juin 2013.

Le churn de Maven est équivalent à celui de Ant, et décroît.
La figure 5.27 nous montre le churn du Build System de JDT-Core.
Comme le montre également le taux de changement, le churn du code Ant évolue par pics souvent forts espacés. Ces pics sont souvent de 100 à 300 lignes modifiées. De juin 2007 à juin 2013, le churn est extrèmement bas, avec seulement quelques pics de 50 à 100 lignes, comme le montrait la période de stabilité dans l'évolution du BLOC. Cela signifie que les changements détectés dans le taux de changement de cette période n'était pour la plupart que des changements de taille très faibles.
Le churn du code Maven quant à lui est de plus en plus petit, malgré que le taux de changement augmente. Les modifications de fin 2012 ne sont en réalité pas grande, seulement une dizaine de lignes. Celles de début 2013 sont de 180 lignes, puis descendent à moins d'une centaine de lignes au mois de juin.

La figure 5.28 nous présente le churn du Build System de JDT-Core par une évolution de Box Plot.
La médiane des pics principaux avant la migration est généralement dans les 20 lignes par commits, et monte parfois jusqu'à 50 lignes. Le troisième quartile monte à plusieurs reprises à 100 lignes par commits.
Après la migration, les pics repérés dans le churn sont, comme le montre le haut taux de changement et les basses valeurs de churn, de petits commits. La médiane est plus proche de la dizaine, et le troisième quartile de la vingtaine de lignes. Seul le premier pic post-migration a un troisième quartile de 100 lignes de code modifiées dans un commit.

Le taux de co-changement de Maven est anecdotique, seulement un pic de 3%.
La figure 5.29 nous montre le couplage entre le code source et le code du Build System de JDT-Core.
Le co-changement est globalement peu élevé, en dessous de 5% des commits la majorité du temps. Il n'y a qu'un seul pic pour Maven, de 3%.

Le co-churn de Maven est presque inexistant, alors que celui de Ant est bien présent malgré des chiffres très bas.
La figure 5.30 et la figure 5.31 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Le code Ant a quelques pics de 50 lignes maximums de commits couplés à du code source. De son coté, le code Maven est totalement indépendant, avec seulement une petite montée de quelques lignes.

Les fichiers de JDT-Core sont souvent tous impactés par les modifications, alors que ceux de Ant ne sont pas beaucoup impactés.
La figure 5.32 nous montre l'impact des changements sur les fichiers du Build System de JDT-Core.
Les changements de Ant s'élèvent au maximum à 40%, mais la majorité du temps les modifications n'impactent pas plus de 20% des fichiers.
Par contre, le Build System Maven est largement plus impacté par les changement, puisque 100% du Build System est changé pendant trois trimestres !
Le Build System Maven est assez petit, il fait seize fichiers, ce qui explique comment il est possible d'atteindre les 100% de fichiers impactés.

En conclusion, les résultats sont globalement positifs. Malgré une augmentation significative du taux de changements avec le Build System Maven, les commits sont de plus petites taille et la taille totale des changements n'est pas plus élevée que celles de Ant. La taille totale du Build System est aussi inférieure à celle de Ant, et le code est presque totalement indépendant du code source.

5.2.5 JDT-UI

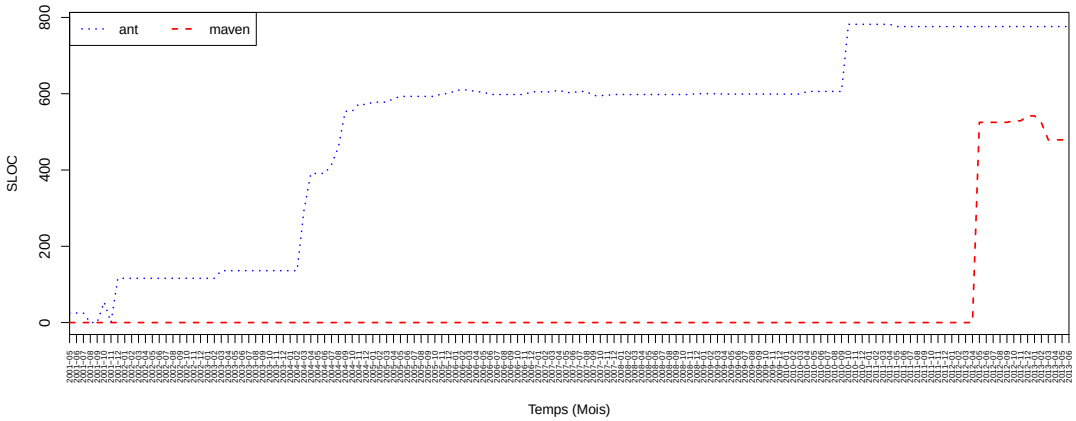

FIGURE 5.33 – BLOC de JDT-UI

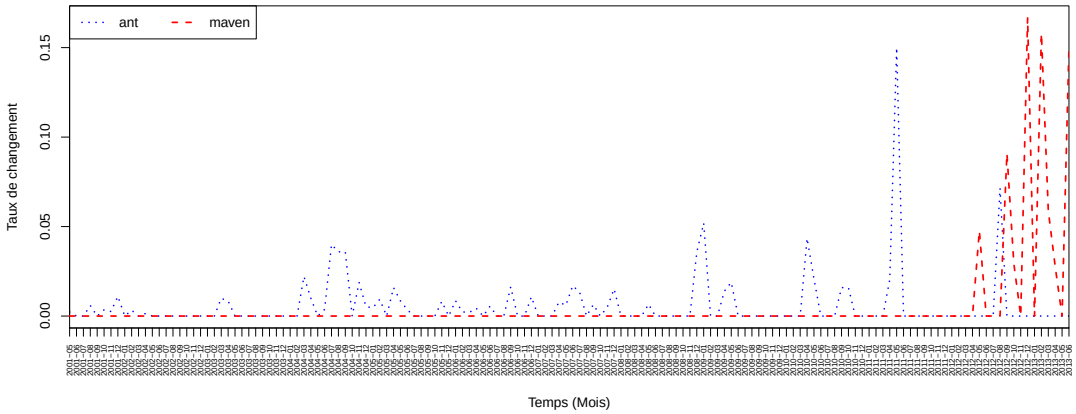

FIGURE 5.34 – Taux de changement de JDT-UI

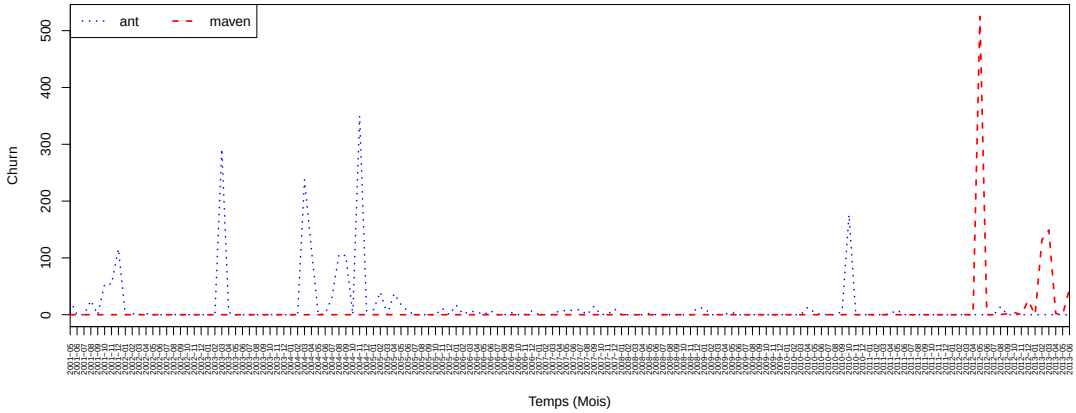

FIGURE 5.35 – Churn de JDT-UI

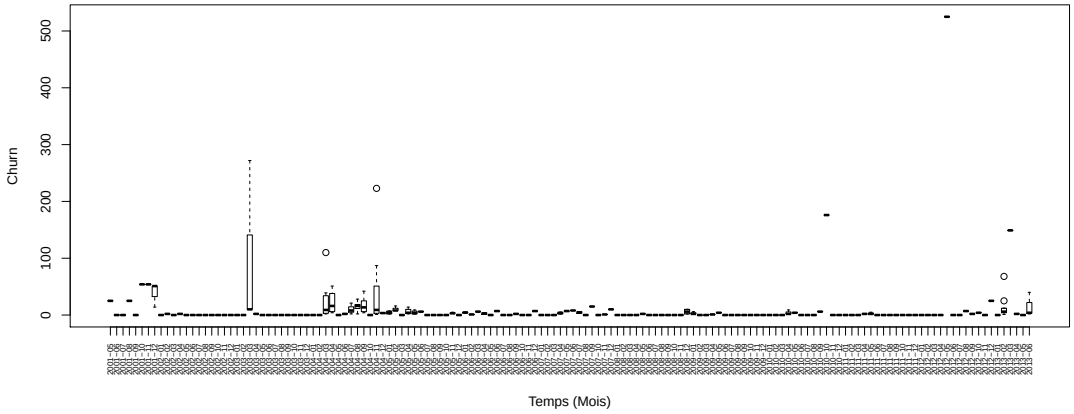

FIGURE 5.36 – Churn de JDT-UI sous forme de Boxplot

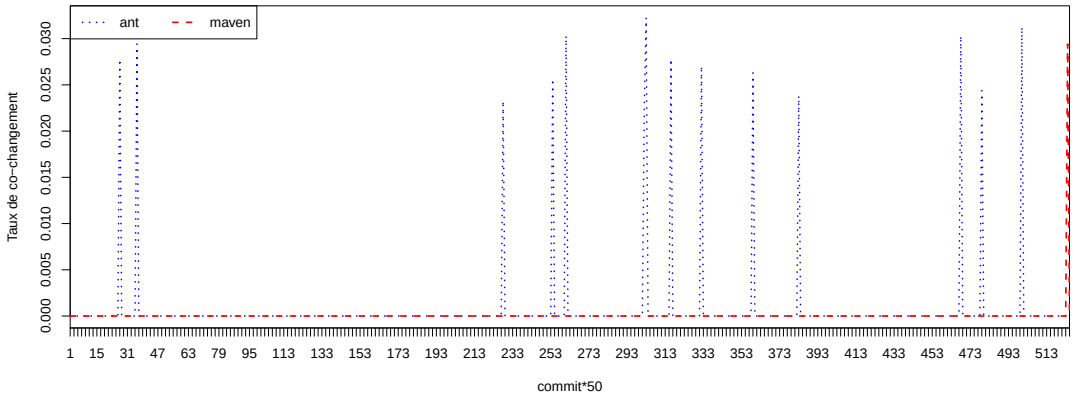

FIGURE 5.37 – Taux de co-changement de JDT-UI

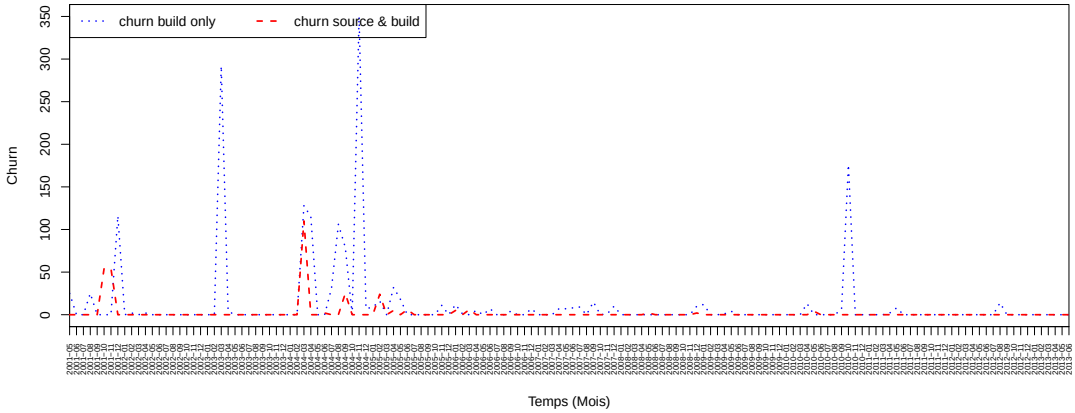

FIGURE 5.38 – Co-churn de Ant de JDT-UI

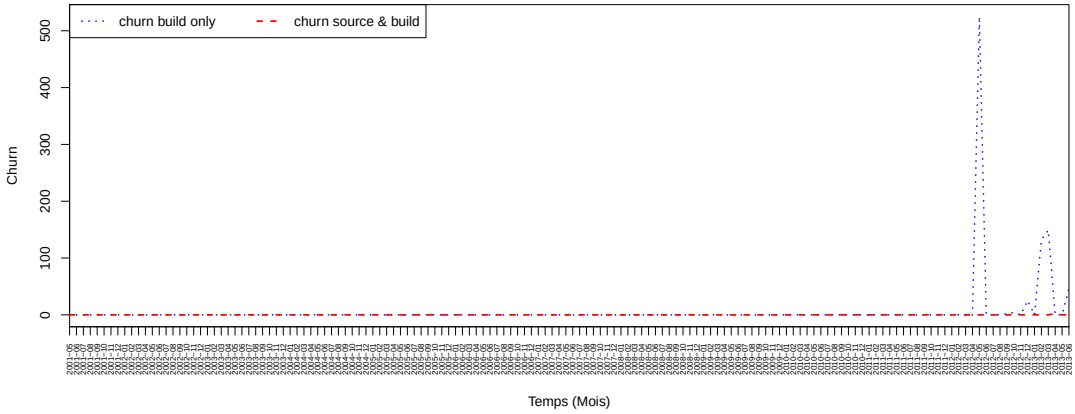

FIGURE 5.39 – Co-churn de Maven de JDT-UI

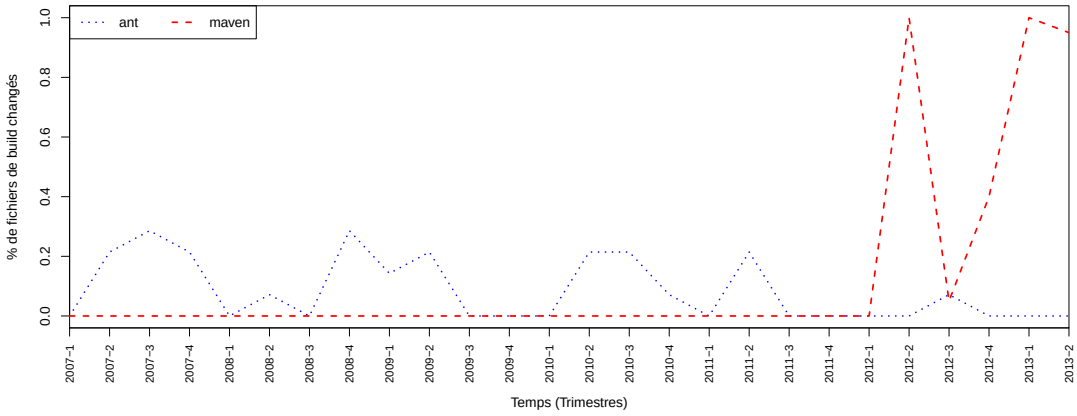

FIGURE 5.40 – Impact sur les fichiers de Build System de JDT-UI

La figure 5.33 nous montre l'évolution du BLOC de JDT-UI.
Le Build System Ant évolue par plateaux, avec trois croissances rapides suivies d'une longue période de stabilité. La migration a lieu en mai 2012. Le Build System Maven reste stable, puis croît légèrement avant de chuter d'une centaine de lignes en mars 2013. Le Build System Maven est de taille inférieure à celui de Ant.

La figure 5.34 nous montre le taux de changement du Build System de JDT-UI.
Le Build System Ant n'est pas souvent modifié, les pics sont très espacés et restent très bas, la majorité n'étant que de 3%.
Le Build System Maven par contre change beaucoup plus fréquemment, avec des pics beaucoups plus élevés ; trois pics s'élèvent à 15%, et un pic à 10%.

La figure 5.35 nous montre le churn du Build System de JDT-UI.
Comme le montre l'évolution du BLOC du code Ant, le churn n'est réellement élevé qu'aux croissances observées. Ces croissances s'élèvent à 300 lignes modifiées. Le code Maven a également un churn très faible, en dehors de la migration et de la chute de mars 2013.

La figure 5.36 nous présente le churn du Build System de JDT-UI par une évolution de Box Plot.
Nous pouvons observer que la plupart des pics de churns sont provoqués par peu de commits, qui ont donc une assez grande taille.

La figure 5.37 nous montre le couplage entre le code source et le code du Build System de JDT-UI.
Les commits couplés au code source sont très rares, aucun pic ne dépassant les 3%. Maven ne connait qu'un seul pic, de 2.8%.

La figure 5.38 et la figure 5.39 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Le co-churn de Ant est négligeable, à l'exception d'un pic de 50 lignes et un de 100 lignes, respectivement en octobre 2001 et en mars 2004. Celui de Maven est presque totalement indépendant du code source, avec seulement quelques lignes couplées.

La figure 5.40 nous montre l'impact des changements sur les fichiers du Build System de JDT-UI.
Les changements de Ant s'élèvent au maximum à 30%, mais il y a de nombreux trimestres où les fichiers ne sont pas modifiés ou presque.
Par contre, le Build System Maven est largement plus impacté par les changement, puisque près de 100% du Build System est changé pendant trois trimestres ! Le Build System Maven est assez petit, il fait vingt fichiers, ce qui explique comment il est possible d'atteindre les 100% de fichiers impactés.

En conclusion, la situation est très proche de son projet-frère, JDT-Core.

5.2.6 Mylyn

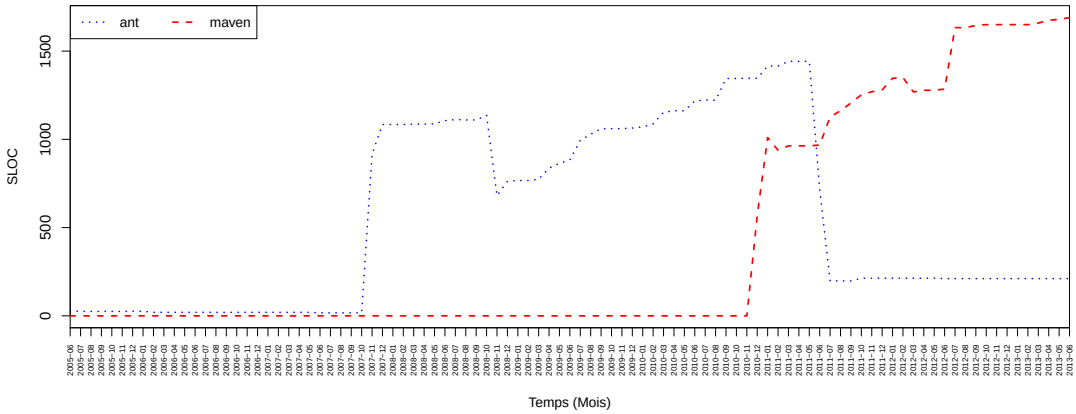

FIGURE 5.41 – BLOC de Mylyn

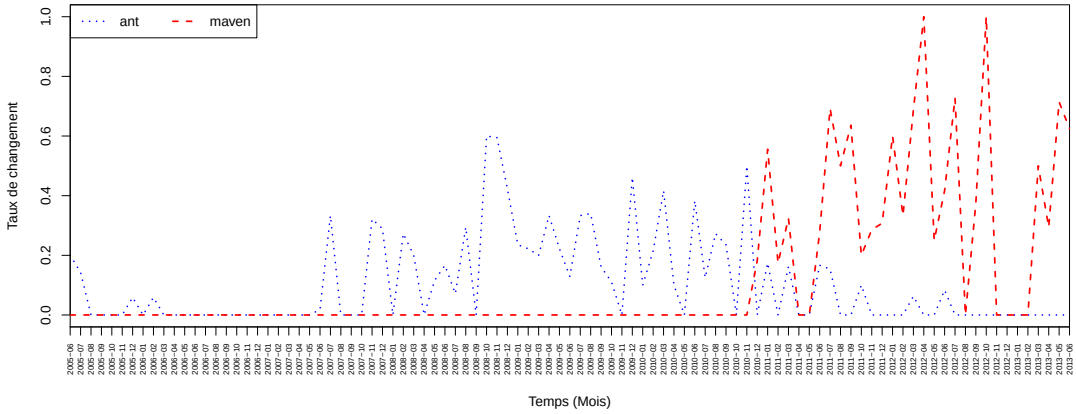

FIGURE 5.42 – Taux de changement de Mylyn

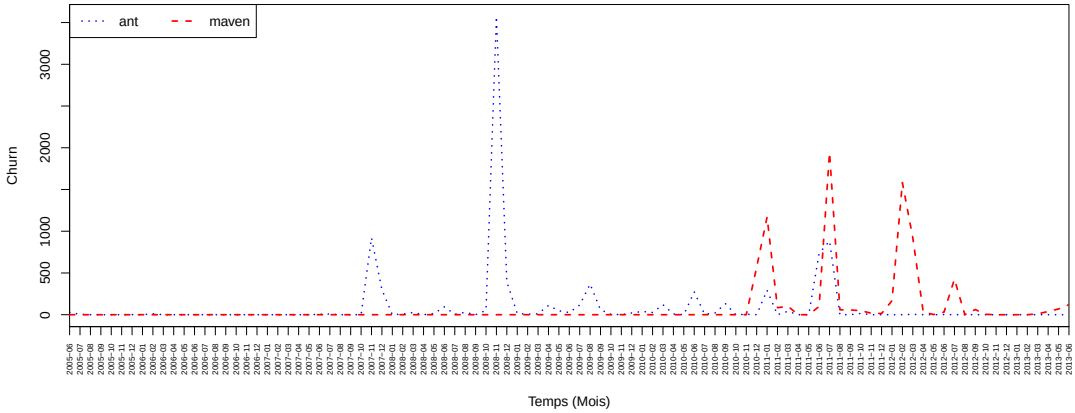

FIGURE 5.43 – Churn de Mylyn

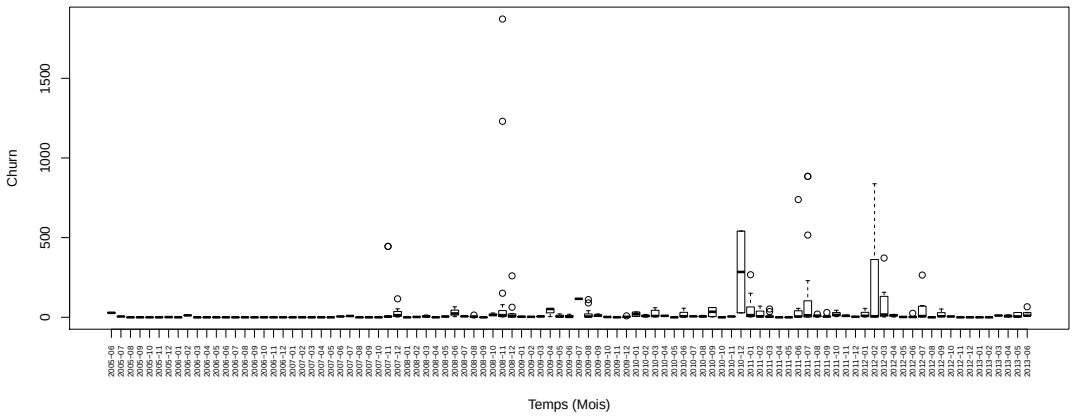

FIGURE 5.44 – Churn de Mylyn sous forme de Boxplot

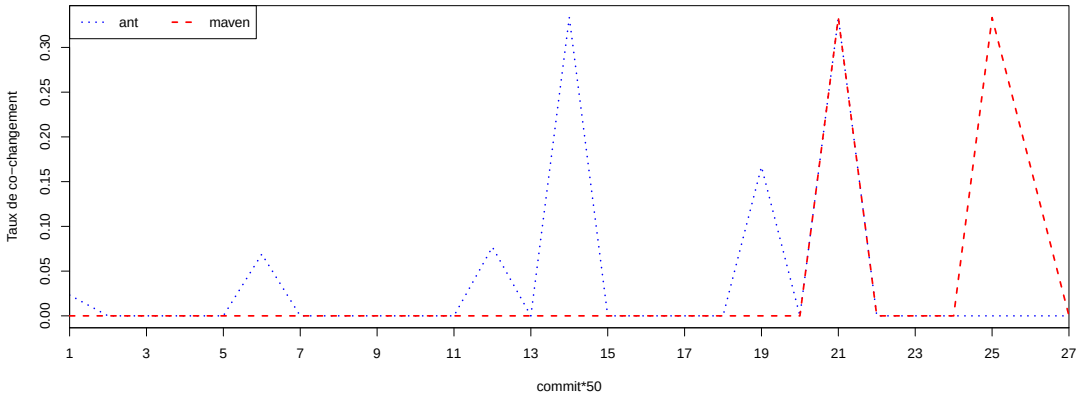

FIGURE 5.45 – Taux de co-changement de Mylyn

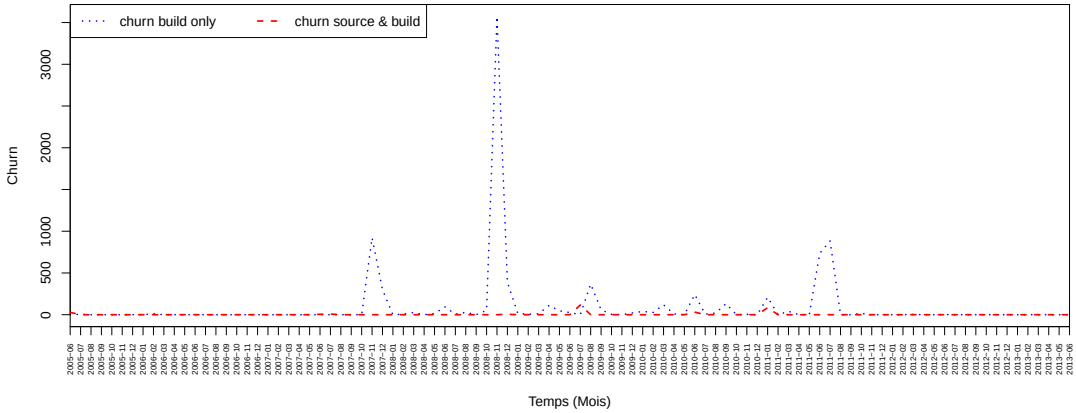

FIGURE 5.46 – Co-churn de Ant de Mylyn

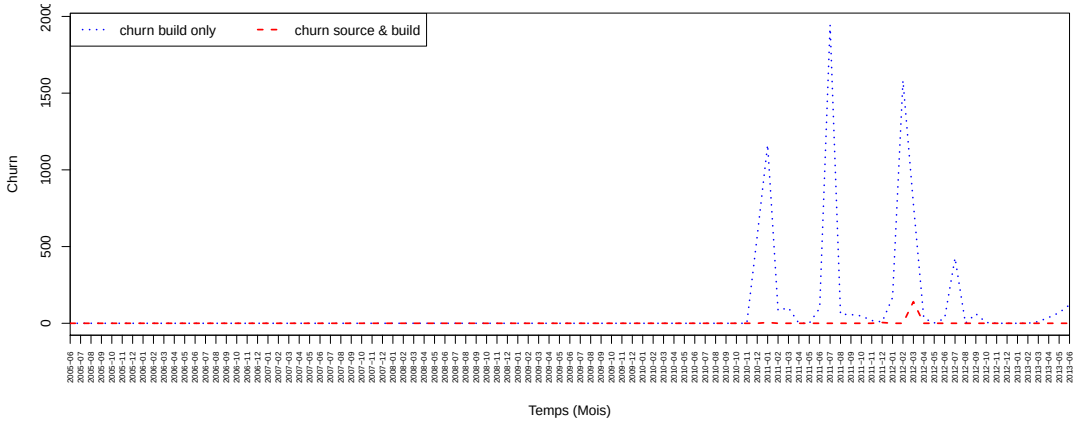

FIGURE 5.47 – Co-churn de Maven de Mylyn

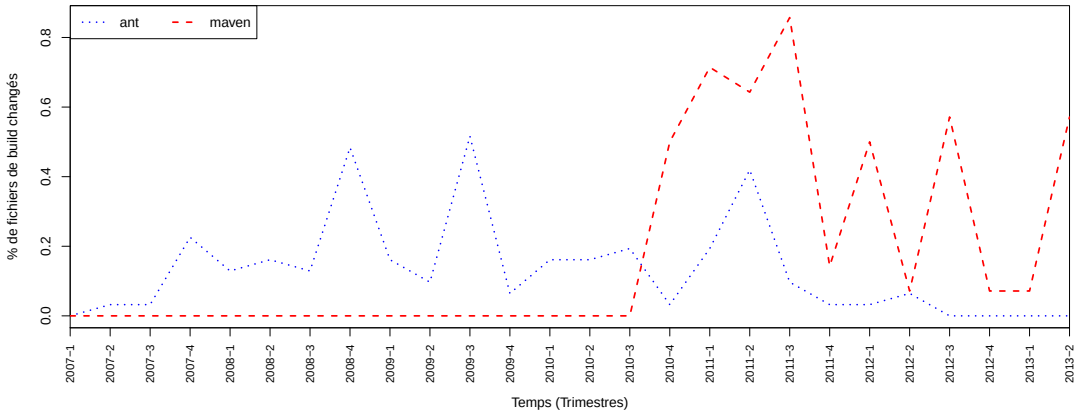

FIGURE 5.48 – Impact sur les fichiers de Build System de Mylyn

Le BLOC de Maven augmentea une croissance soutenue, puis une période de stabilité la dernière année.
La figure 5.41 nous montre l'évolution du BLOC de Mylyn.
Le Build System Ant n'évolue pas du tout pendant les deux premières années de développement, l'équipe de développement se contentant du Build System automatique d'Eclipse. En octobre 2011, le Build System croît d'un coup jusqu'à 1200 lignes. Le Build System reste ensuite stable jusqu'en octobre 2008, où il chute de 500 lignes. Cette chute est causée par de nombreuses corrections de bugs et une modification du Build System pour supporter l'intégration continue.
Ensuite, la croissance devient linéaire, montant progressivement jusqu'à 1500 lignes en mai 2011. 6 mois après la migration, le Build System Ant chute brusquement, jusqu'à 200 lignes au total. Ensuite, il ne varie presque plus.
La migration vers le Build System Maven s'est déroulée en décembre 2010 par un big bang. le Build System monte progressivement, avec trois périodes de stabilité ; pendant 5 mois après la migration, de mars à juin 2012 et d'août 2012 à juin 2013. Le Build System est monté d'un millier de lignes de code à la migration jusqu'à 1600 lignes en juin 2013.

Le taux de changement de Maven est beaucoup plus élevé que celui de Ant.
La figure 5.42 nous montre le taux de changement du Build System de Mylyn.
Le code Ant varie beaucoup, de 30 à 50% la majorité du temps, avec un pic à 60% en octobre 2010. Après la migration, les changements sont beaucoup plus rares et de faible pourcentage.
Le code Maven est encore plus modifié, de 30% à 70%, avec des pics jusqu'à 95%.

Le code Maven et Ant ont tous les deux connus trois mois de grands changements. En dehors de ça, les changement sont de taille similaire.
La figure 5.43 nous montre le churn du Build System de Mylyn.
Les modifications des deux Build Systems sont de taille modérée la plupart du temps, de quelques centaines de lignes. Pendant les périodes hors-pics, le churn de Maven est significativement plus faible.
Il y a cependant six pics avec un très grand churn, pas toujours visibles dans l'évolution du BLOC : trois pics de Ant et trois pics de Maven.

Le premier pic de Ant est causé par l'ajout du Build System complet en octobre 2011, avec 1000 lignes ajoutées. Le deuxième pic sont les changements qui ont causé la chute de 500 lignes, avec les corrections de bugs et le passage à l'intégration continue. Cette transition a été difficile à réaliser, comme le montre le haut taux de changements, 60% et les 3500 lignes modifiées dans cette période.
Le dernier pic est la chute du BLOC Ant après la migration, d'un millier de lignes.

Le premier pic de Maven est bien entendu causé par la migration, avec 1200 lignes ajoutées.
Le deuxième pic en juillet 2011 de 1600 lignes est causé par la préparation pour la release 3.6 de Mylyn. Le Build System a donc subit une restructuration et des debugs.
Le dernier pic de 1500 lignes en février 2012 est causé par des debugs.

La figure 5.44 nous présente le churn du Build System de Mylyn par une évolution de Box Plot.
Nous observons que tous les pics du churn ont été causés par des commits de très grande taille, une grosse partie d'entre eux étant supérieurs à 500 lignes modifiées. Les commits sont aussi grands avant la migration qu'après.

La figure 5.45 nous montre le couplage entre le code source et le code du Build System de Mylyn.
Aussi bien Ant que Maven ont quelques pics de commits couplés. Ant a deux pics qui s'élèvent à 30%, tout comme Maven.

La figure 5.46 et la figure 5.47 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Les commits couplés n'ont par contre que peu de churn, aussi bien pour Ant que Maven.

Les fichiers de Maven sont beaucoup plus impactés que ceux de Ant.
La figure 5.48 nous montre l'impact des changements sur les fichiers du Build System de Mylyn.
La maintenance du Build System Ant touche en général autour de 20% des fichiers, et jusqu'à 50% à deux reprises. Après la migration, il y a encore un pic à 40% lors du trimestre ayant vu la suppression de la plupart du Build System Ant, puis presque plus de fichiers sont modifiés.
La maintenance du Build System Maven touche globalement beaucoup plus de fichiers que Ant, avec des pics dépassant régulièrement les 40% de fichiers touchés, et un pic monte jusqu'à 80% des fichiers.

En conclusion, les données montrent que le Build System est beaucoup plus suceptible de changer. En dehors des mois concentré sur la correction de bugs et la restructuration du Build System, le churn de Maven est similaire à celui de Ant.
Concernant le couplage, il n'y a pas eu beaucoup d'amélioration, le couplage étant déjà faible avec le Build System Ant.
De plus, le développeur à qui nous avons posé nos questions nous a informé que l'objectif en migrant vers Maven était d'avoir un Build System plus facile à comprendre, pour que toute l'équipe puisse facilement le modifier. Cet objectif semble être réussi, comme le montre le taux de changement.
La migration ne semble donc pas être une réussite pour la diminution du travail des développeurs, mais bien une réussite pour l'objectif de l'équipe de développement.

5.2.7 PDE

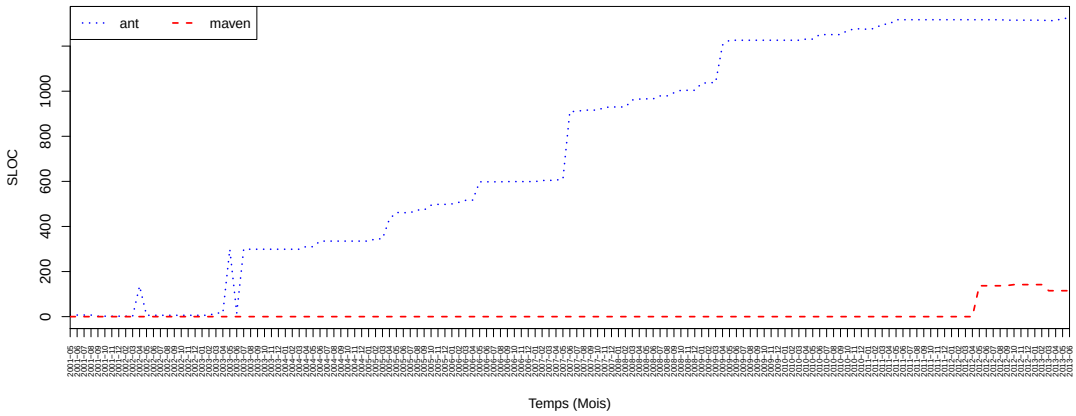

FIGURE 5.49 – BLOC de PDE

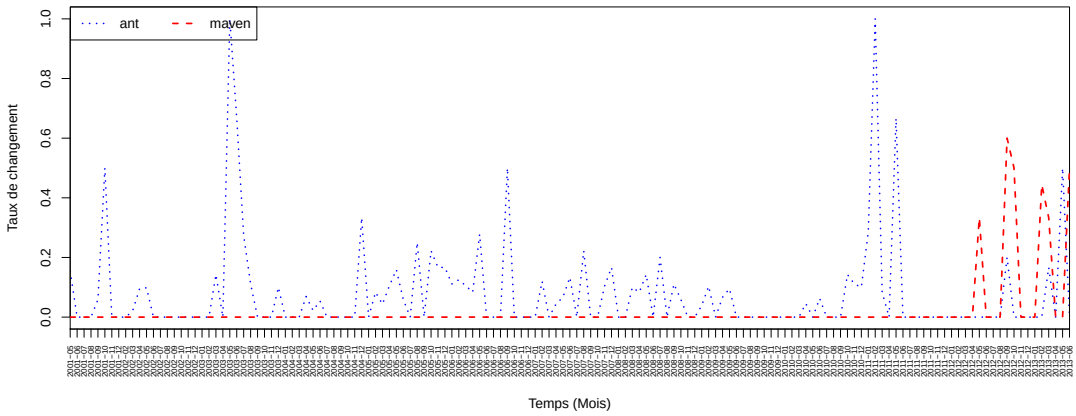

FIGURE 5.50 – Taux de changement de PDE

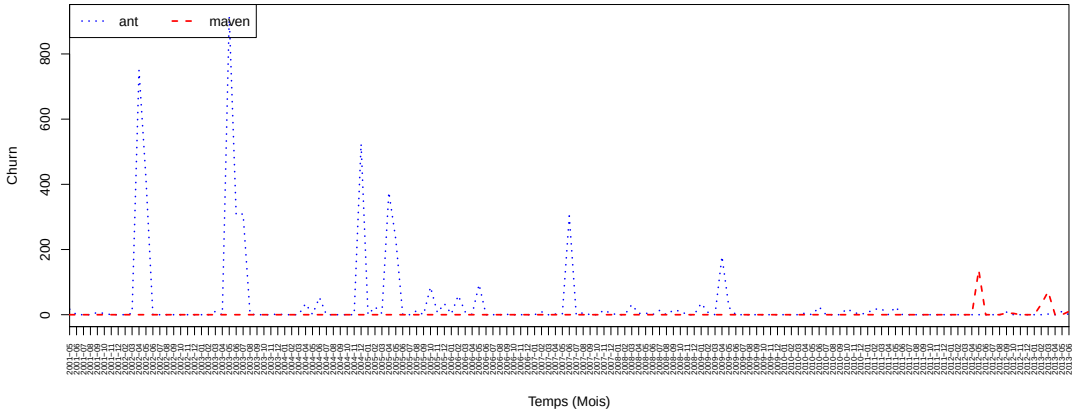

FIGURE 5.51 – Churn de PDE

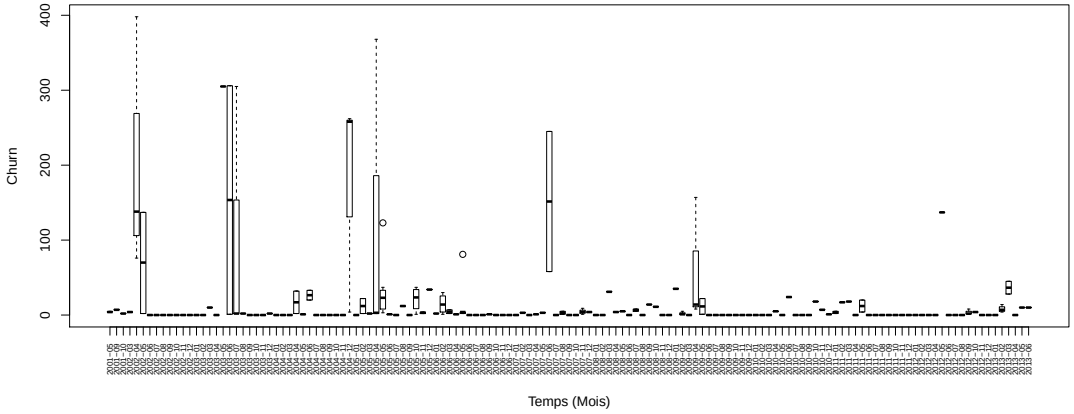

FIGURE 5.52 – Churn de PDE sous forme de Boxplot

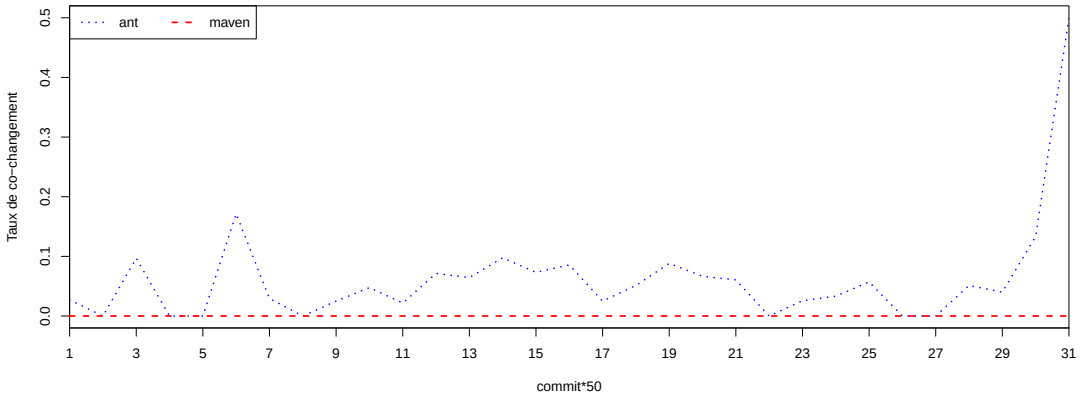

FIGURE 5.53 – Taux de co-changement de PDE

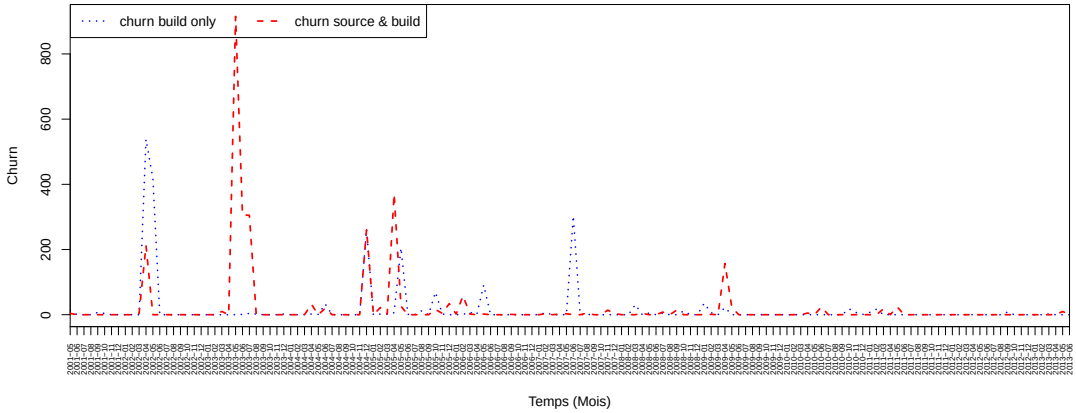

FIGURE 5.54 – Co-churn de Ant de PDE

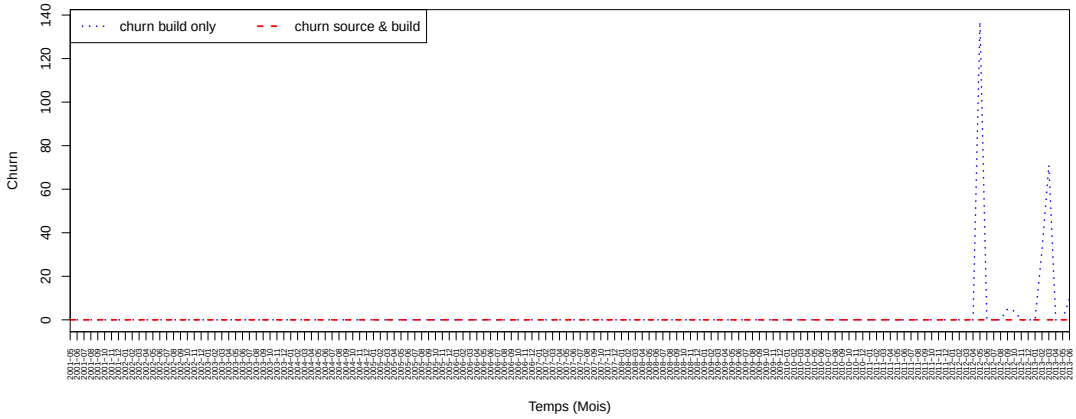

FIGURE 5.55 – Co-churn de Maven de PDE

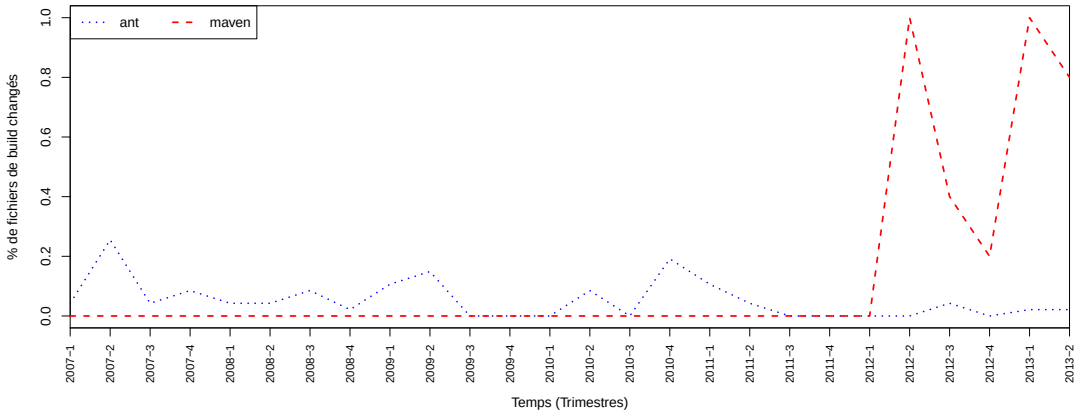

FIGURE 5.56 – Impact sur les fichiers de Build System de PDE

Le BLOC de Maven est très faible et stable.
La figure 5.49 nous montre l'évolution du BLOC de PDE.
Le Build System Ant a une croissance par plateau, avec cinq croissances brusques suivie d'une période de stabilité ou de faible croissance linéaire. De mai 2009 à juin 2013, le Build System reste relativement stable.
La migration a été réalisée en mai 2012 par un big bang, et le code Maven a peu évolué depuis, avec seulement une chute notable que quelques dizaines de lignes. Sa taille fait moins de 200 lignes.

La figure 5.50 nous montre le taux de changement du Build System de PDE.
Le Build System Ant évolue souvent avec un pourcentage entre 15 et 30%, et connait deux pics proches de par une évolution de Box Plot en mai 2003 et février 2011.
Le Build System Maven n'a connu que quatre pics, mais ceux ci sont entre 30 et 60%.

Le code Maven a très peu de churn. La figure 5.51 nous montre le churn du Build System de PDE.
Le churn de Ant se fait par grands pics suivit par de longues périodes de faible taille. Cela concorde avec l'évolution en plateau observée dans l'évolution du BLOC. Le pic de décembre 2004 est assez surprenant, car il est de 550 lignes mais le BLOC ne change pas du tout ce mois là. En réalité, il s'agit d'un fichier build.xml de grande taille qui a été ajouté, pour finalement être supprimé deux commits après, soit deux heures plus tard.
Le churn de Maven n'a que deux vrais pics, la migration et la légère chute de BLOC.

La figure 5.52 nous présente le churn du Build System de PDE par une évolution de Box Plot.
Les commits de Ant sont souvent de très grandes taille par rapport au BLOC total du Build System, comme le montre les médianes assez élevées, de 100 à 300 lignes modifiées régulièrement.
La migration s'est faite en un seul commit, et les changements qui ont causés la chute de BLOC de Maven a été réalisé en trois commits d'environ 50 lignes chacun.

Maven est totalement indépendant du code source La figure 5.53 nous montre le couplage entre le code source et le code du Build System de PDE.
Le Build System Ant a en moyenne une dizaine de pourcentage de commits couplés. Sur les derniers commits, 50% étaient couplés. Par contre le code Maven est totalement indépendant.

La figure 5.54 et la figure 5.55 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Le churn des commits couplés concernant le code Ant est très faible, à l'exception d'un pic à 50 lignes, soit tout le churn des mois d'octobre et novembre 2001 et un pic à 100 lignes, soit la moitié du churn du mois de mars 2004.
Comme l'indiquait le taux de co-changement, le code Maven est totalement indépendant, avec aucun churn couplé.

La figure 5.56 nous montre l'impact des changements sur les fichiers du Build System de PDE.
Les fichiers Ant sont faiblement impactés par les changements, moins de 25% des fichiers sont modifiés chaque trimestre. Par contre, les fichiers Maven sont bien plus impactés, avec un pic normal de 100% à la migration, mais également un autre pic à 100% au premier trimestre 2013. Le plus faible pourcentage de fichiers Maven modifié est de 25%, soit le maximum des fichiers Ant.
Ceci peut être facilement expliquer par le fait que le Build System Maven n'est constitué que de cinq fichiers, alors que celui de Ant est constitué de 47 fichiers.

5.2.8 JBossTools

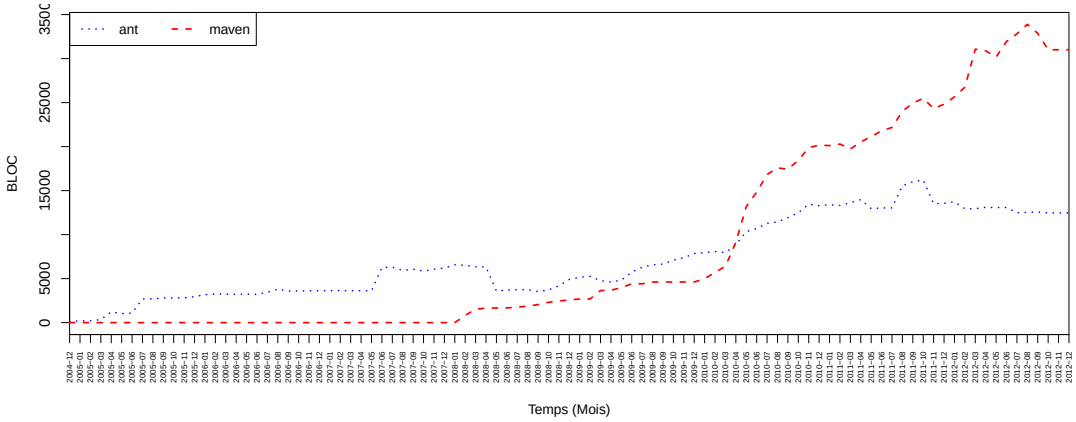

FIGURE 5.57 – BLOC de JBossTools

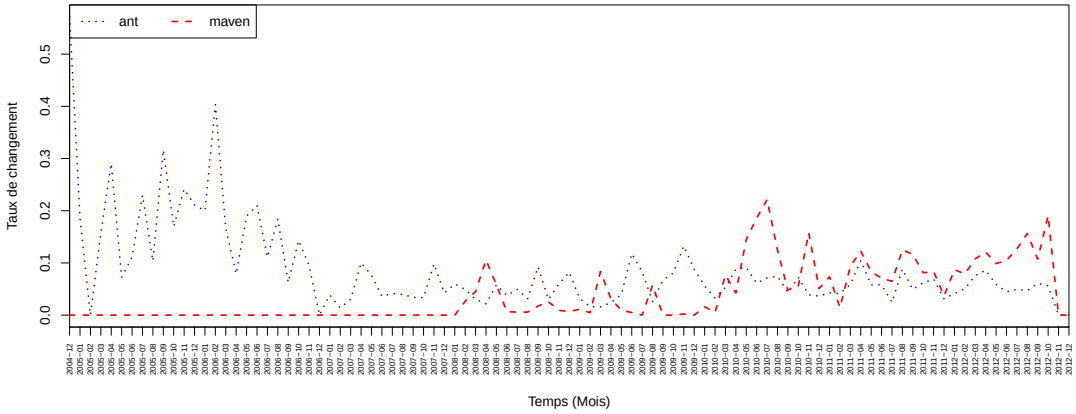

FIGURE 5.58 – Taux de changement de JBossTools

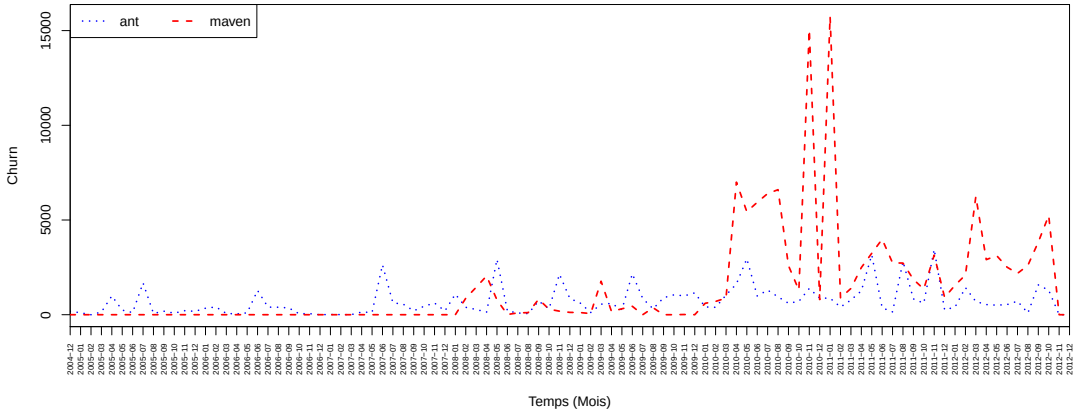

FIGURE 5.59 – Churn de JBossTools

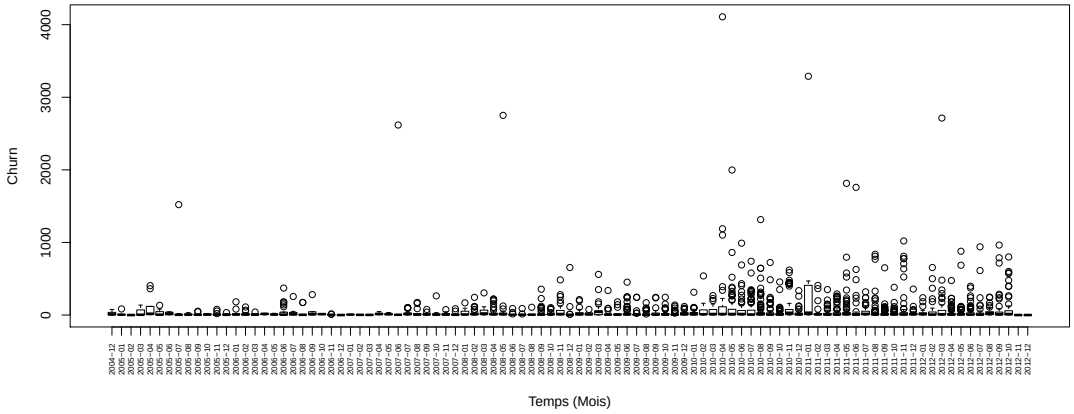

FIGURE 5.60 – Churn de JBossTools sous forme de Boxplot

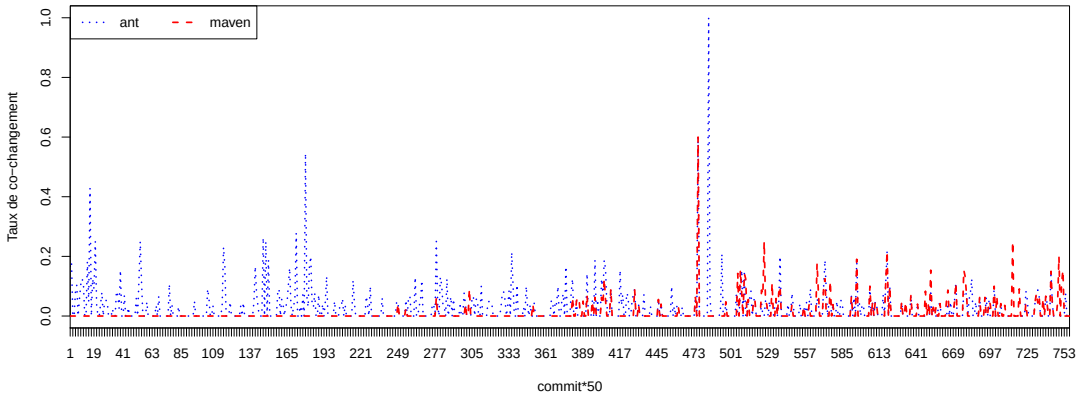

FIGURE 5.61 – Taux de co-changement de JBossTools

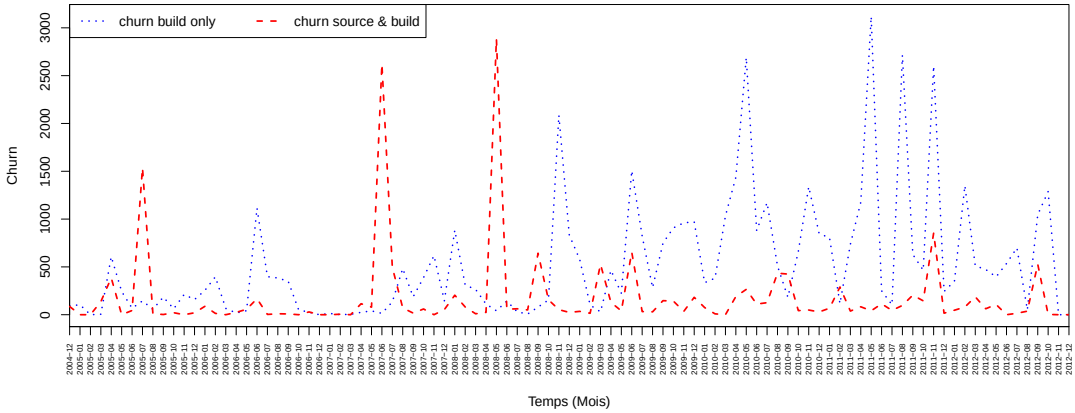

FIGURE 5.62 – Co-churn de Ant de JBossTools

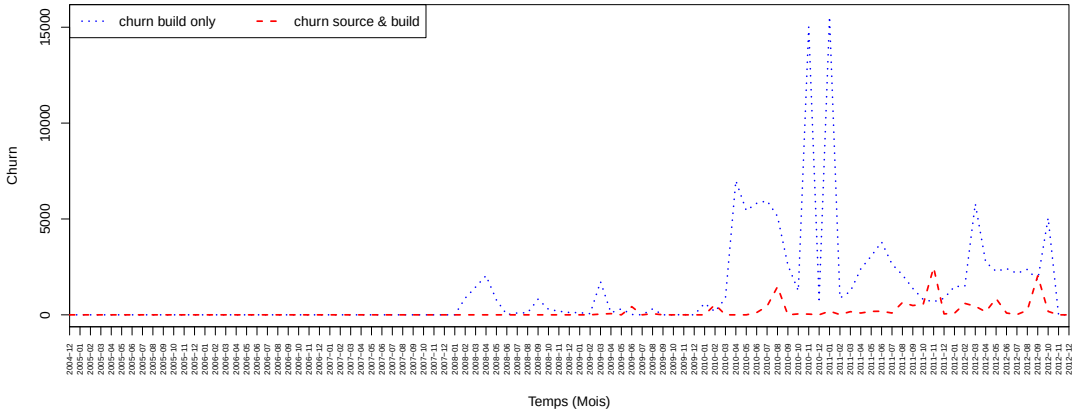

FIGURE 5.63 – Co-churn de Maven de JBossTools

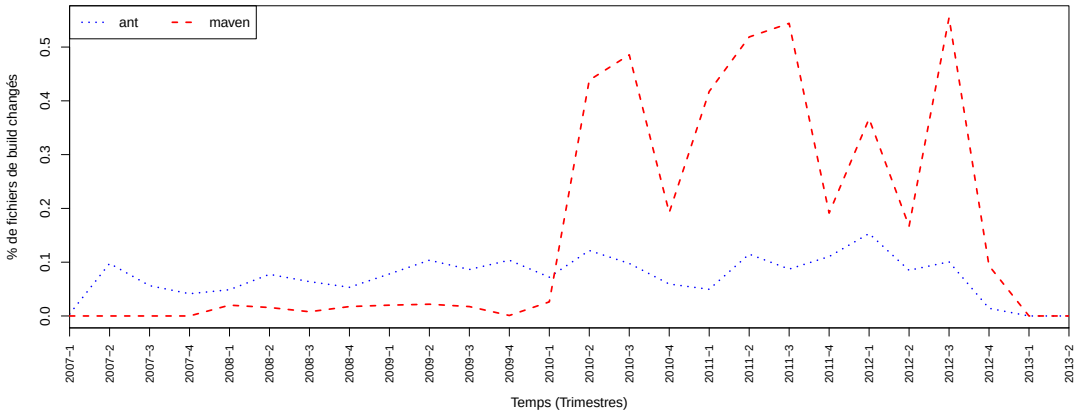

FIGURE 5.64 – Impact sur les fichiers de Build System de JBossTools

Evolution progressive de Maven, qui atteint une très grande taille.
La figure 5.57 nous montre l'évolution du BLOC de JBossTools.
L'évolution du Build System de JBossTools est particulière. L'équipe de développement a décidé d'utiliser Tycho dès l'apparition de cet outil. Ceci explique l'évolution progressive du Build System Maven, bien différente des autres projets.

Le Build System Ant évolue de façon linéaire, atteignant les 6000 lignes de BLOC au moment de la migration, en février 2008. Dû à l'adoption précoce de Tycho, beaucoup de fonctionnalités manquaient, de ce fait le Build System Ant a continué à être développé pour contourner les limitations de Tycho. Le BLOC continue donc d'évoluer jusqu'en octobre 2011, où il culmine à 16000 lignes. Ensuite sa taille ne fait plus que décroire, jusqu'à 12000 lignes en décembre 2012.

Le Build System Maven débute son développement en février 2008. L'évolution est d'abord assez lente, ne montant qu'à 6000 lignes en mars 2010. A cette période, le plug-in Tycho est officiellement proposé à la communauté Eclipse. Le développement du Build System Maven décolle alors, et il monte de 12000 lignes en quatre mois, atteignant alors les 18000 lignes en juillet 2010. La croissance ralenti ensuite, tout en restant assez rapide. Le code Maven culmine à 34000 lignes en août 2012.

Taux de changement de Maven inférieur à celui des deux premières années de Ant. Le code Ant reste constamment maintenanu pendant tout le développement.
La figure 5.58 nous montre le taux de changement du Build System de JBossTools.
Le Build System Ant change le plus fréquemment au début du développement, dépassant les 30% de commits à trois reprises. Le développement se calme décembre 2006. A partir de 2007, le taux de changement du code Ant reste dans les alentours de 10% pour ses plus hautes valeurs.

Le Build System Maven débute avec un taux de changement très faible, de quelques pourcents avec trois pics inférieur à 10% jusqu'en mars 2010. Ensuite, lors des quatre mois de forte croissance du BLOC, le taux de changement monte jusqu'à 25%.
Ensuite, le taux de changement reste constamment entre 10 et 20% des commits.

Le churn de Maven est très important, avec deux pics à 15000 lignes et de nombreux mois à 6000.
La figure 5.59 nous montre le churn du Build System de JBossTools.
Le churn de Ant reste proportionnellement assez faible, ne dépassant jamais les 3500 lignes modifiées. La période pré-migration est la plus calme, avec des périodes de très faible churn. La plupart des mois ont un churn entre 100 et 400 lignes, et les pics s'élèvent à 1700 lignes pour le plus haut en juin 2007. Ensuite, il y a fréquemment des commits dépassant les 1000 lignes modifiées.

Le churn de Maven est quant à lui bien plus élevé. La période avant mai 2010 ressemble à celle de Ant, avec une majorité de mois avec un faible churn, de 100 à 400 lignes, et quelques pics jusqu'à 2000 lignes. Ensuite, la période de forte croissance de avril 2010 à juillet 2010 observée dans le BLOC se traduit par quatre mois à 6000 lignes modifiées. En novembre 2010 et février 2011, il y a deux pics impressionnants de 15000 lignes. Le premier pic est dû à une modification du "jdocbook" de 440 lignes, qui est répercutée sur plus d'une vingtaine de fichiers POM dans des commits distincts. Le pic de janvier est présent pour la même raison, des "mises à jours générales" de 400 lignes sur plus d'une vingtaine de fichiers pom.xml.
Ensuite, le churn reste entre 1200 et 4000 lignes modifiées, avec un pic à 6000 en mars 2012.

La figure 5.60 nous présente le churn du Build System de JBossTools par une évolution de Box Plot.
Les boxplot nous montrent que de très nombreux de commits approchent les 1000 lignes modifiées, et beaucoup les dépassent largement, atteignant même 4000 lignes en avril 2010. Ce commit était une modification de nombreux POM pour passer la génération de la documentation avec Maven 3.0.

Aucun changement entre Ant et Maven pour le taux de co-changement.
La figure 5.61 nous montre le couplage entre le code source et le code du Build System de JBossTools.
Le taux co-changement est autour de 20% pendant toute la durée du développement, aussi bien pour Ant que Maven. Ce taux est stable, et n'a pas évolué après la migration. Cette valeur est assez élevée, mais reste raisonnable.

Le co-churn de Maven est plus bas que celui de Ant, avec pics moins nombreux de plus petite taille.
La figure 5.62 et la figure 5.63 nous montrent le churn des commits couplés au code source comparé au churn des commits ne touchant qu'au Build System respectivement de Ant et de Maven.
Le co-churn de Ant est constamment présent, et fait plusieurs pics assez élevés, jusqu'à 3000 lignes. Les trois pics les plus élevés sont d'ailleurs presque les seuls churns de leur mois.
En comparaison, le co-churn de Maven est beaucoup plus faible que le churn indépendant. Les plus hauts pics montent à 2500 lignes.

Les fichiers de Maven sont largement plus impactés que ceux de Ant.
La figure 5.64 nous montre l'impact des changements sur les fichiers du Build System de JBossTools.
Il y a constamment entre 5 et 15% de fichiers ant impactés par la maintenance du Build System. C'est beaucoup plus faible que les résultats de Maven, où jusqu'à 50% des fichiers sont touchés.

En conclusion, la migration n'a pas été parfaite, dû à l'aide au développement de Tycho qu'à fournit l'équipe de JBossTools. La migration a été progressive mais n'a pas donné un Build System réellement plus stable. Le taux de changement et de co-changement sont restés similaires à celui de Ant, et le churn a beaucoup augmenté.

5.3 Conclusion des résultats

Dans cette section, nous allons rassembler les résultats obtenus dimension par dimension, pour évaluer la tendance générale dans chacune d'entre elles suite à la migration.

5.3.1 Evolution de la taille du Build System

	Evolution du BLOC de Maven	**Conclusion**
LinuxTools	Dernière année plus stable	**V**
CDT	Plus petit, évolution lente	**V**
BPEL	Moins stable pendant une année	**X**
JDT-Core	Plus petit, croissances brusques	~
JDT-UI	Plus petit, décroît	**V**
Mylyn	Croît autant que Ant, dernière année stable	**V**
PDE	Beaucoup plus petit, stable	**V**
JBossTools	Très forte croissance	**X**

Comme observé dans le papier de McIntosh *et al.*[5], nous avons également constaté que les Build System, aussi bien Ant que Maven, évoluent bel et bien avec le temps.
De nombreux projets continuent à développer le Build System Ant après la migration, car certaines fonctionnalités sont absentes de Maven ou Tycho et doivent donc toujours être utilisées via le code Ant. Celui-ci doit donc naturellement continuer à être maintenu. Mais les deux Build System ne cohabitent pas, le Build System Maven remplace entièrement celui de Ant, il utilise juste des parties dont les spécificités ne peuvent pas être formulées avec Maven.

Pour les projets de moins de 1500 lignes de BLOC, tels PDE et les deux répertoires de JDT, le Build System Maven reste assez simple, et a donc une taille largement inférieure à celle du Build System Ant.
Mais lorsque le projet est de taille plus importante, son Build System est naturellement plus complexe, et Maven a donc besoin de configurations personnalisées et des fonctionnalités supplémentaires. De ce fait, le Build System Maven fait au moins autant de lignes que le Build System Ant, et parfois beaucoup plus, comme c'est le cas avec CDT, LinuxTools et JBoss Tools. La raison étant que Maven est très verbeux lorsqu'il faut s'écarter de sa configuration par défaut.

5.3.2 Modifications du code dans le Build System

	Taux de changement	Churn	Conclusion
LinuxTools	Légèrement inférieur	Supérieur	~
CDT	Inférieur	Largement inférieur	V
BPEL	Supérieur	Supérieur	X
JDT-Core	Supérieur et croissant	Inférieur	V
JDT-UI	Supérieur et croissant	Inférieur	V
Mylyn	Supérieur	Inférieur	V
PDE	Moins fréquent	Largement inférieur	V
JBossTools	Inférieur puis similaire	Supérieuré	~

Le Build System Maven réclame globalement des mises à jour plus fréquentes que le Build System Ant qui le précédait. Mais son churn, quant à lui, malgré un taux de changement plus élevé en moyenne, est souvent plus faible.
Cela montre que bien que le Build System réclame plus de maintenance, cependant ces dernières sont au final bien plus réduites et donc rapide à faire.
L'analyse manuelle des commits a effectivement montré que de nombreux changements sont de simples mises à jour de numéros de version de Maven, de Tycho ou d'un autre plug-in. Ces changements doivent souvent être répercutés dans de nombreux fichiers POM. Ils sont parfois effectués sur plusieurs commits.

Voici un exemple récurrent de modification mettant à jour un plug-in. Dans cet exemple, le plug-in LTTNg est mis à jour de la version 2.0.0 vers la nouvelle version 2.1.0.

```
<parent>
  <artifactId>linuxtools-lttng-parent</artifactId>
  <groupId>org.eclipse.linuxtools.lttng</groupId>
 - <version>2.0.0-SNAPSHOT</version>
 + <version>2.1.0-SNAPSHOT</version>
</parent>\\
```

La migration donne donc souvent des résultats positifs pour cette dimension. Bien qu'il y ait plus de commits que précédemment, ceux-ci ont une taille beaucoup plus faible.

Couplage du code du Build System avec le code source

	Taux de co-changement	**Co-Churn**	**Conclusion**
LinuxTools	Légèrement inférieur	Légèrement inférieur	∼
CDT	Inférieur	Largement inférieur	**V**
BPEL	Similaire	Beaucoup inférieur	**V**
JDT-Core	Très faible	Presque aucun	**V**
JDT-UI	Très faible	Presque aucun	**V**
Mylyn	Similaire	Similaire (très bas)	∼
PDE	Aucun	Aucun	**V**
JBossTools	Similaire	Similaire	∼

Pour le Build System Ant de tous les projets, le couplage du code du Build System avec le code source était présent. Le taux de changement atteignait et dépassait dans presque tous les cas les 25% des commits modifiant le Build System au moins une fois. Ce chiffre montait même à 60% pour LinuxTools. Le co-churn était donc lui aussi présent, avec des pics de taille significative pour de nombreux projets.

Par contre la situation s'améliore fortement après la migration avec le Build System Maven. La différence dans le taux de co-changement n'est pas toujours significative, mais le co-churn lui a drastiquement baissé, au point d'être parfois inexistant ou presque.

Migrer vers Maven offre donc bel et bien un Build System plus flexible, qui nécessite beaucoup moins de maintenance directe lorsque le code source est changé.

5.3.3 Impact sur les fichiers

La maintenance du Build System Maven impacte significativement plus de fichiers pour tous les projets que la maintenance de leur ancien Build System Ant. Ces pourcentages élevés de fichiers modifiés lors des maintenances sont dus à la structure même de Maven. Beaucoup de fichiers pom.xml se ressemblent et certaines modifications dans un fichier, telle une mise à jour d'un plug-in, doivent être répercutées sur plusieurs autres fichiers.
De plus, le Build System Maven est souvent constitué de moins de fichiers, ce qui augmente les pourcentages obtenus.

Concernant les Treemaps, les tendances sont similaires entre tous les projets. Dans un soucis de concision, nous ne montrerons que les années les plus significatives de LinuxTools, un projet de taille suffisamment importantes où les résultats restent les plus lisibles en version papier.

Couleur de la case	**Nombre de commits**
Blanc	0
Vert	1-9
Jaune / Orange	10+
Rouge	Max

TABLE 5.1 – Coloration des cases de la Treemap en fonction du nombre de commits.

En 2009 (figure 5.65) et 2010 (figure 5.66), qui sont pourtant les deux années où Ant a le plus évolué, la partie impactée par les changements est très faible. Le fichier principal du Build System est le build.xml du dossier releng, soit Release Engineering. La très grosse majorité des changements ont été effectués dans ce fichier, et les autres fichiers n'ont eu que très peu de commits les ayant changés.

En 2011 (figure 5.67) et 2012 (figure 5.68), le Build System Maven a, quant à lui, été énormément changé. En 2011 se déroulait la migration, il est donc normal que beaucoup de fichiers soient touchés, vu qu'ils ont été créés cette année là. Mais la tendance se confirme en 2012, où autant de fichiers ont été modifiés, mais beaucoup moins de fois qu'en 2011.
Le pom.xml situé à la racine est celui qui a reçu le plus de changements. Le reste est également très faiblement touché.

Les modifications de Maven sont donc beaucoup plus distribuées, elles touchent chaque année une très grande partie du Build System. Mais ces changements sont effectués de manière semblables à Ant, avec un fichier principal très souvent changé, et les autres peu changés.

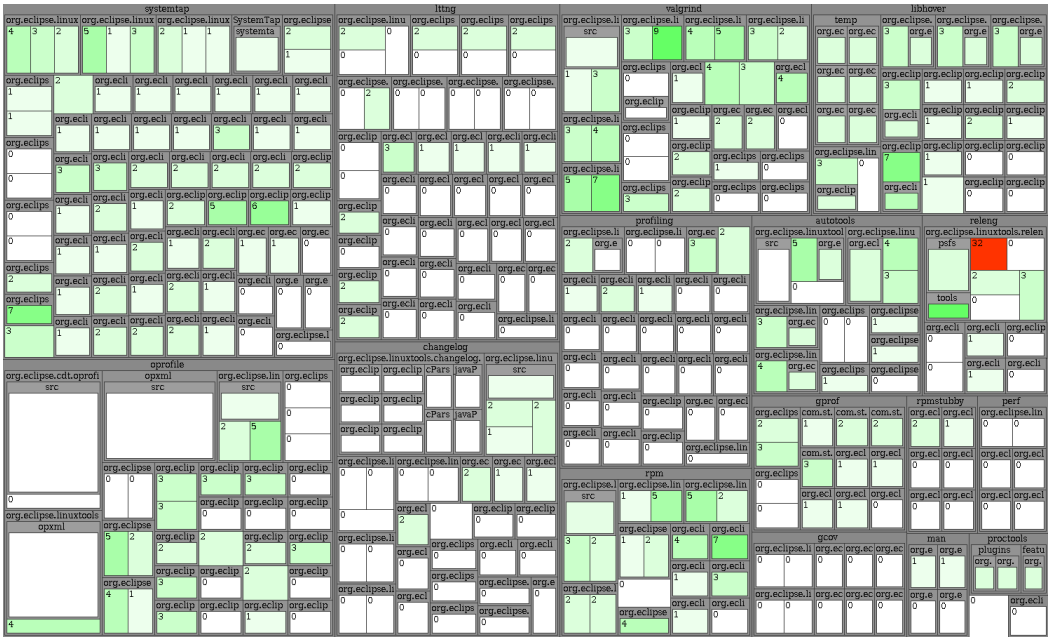

FIGURE 5.65 – Treemap de Ant en 2009

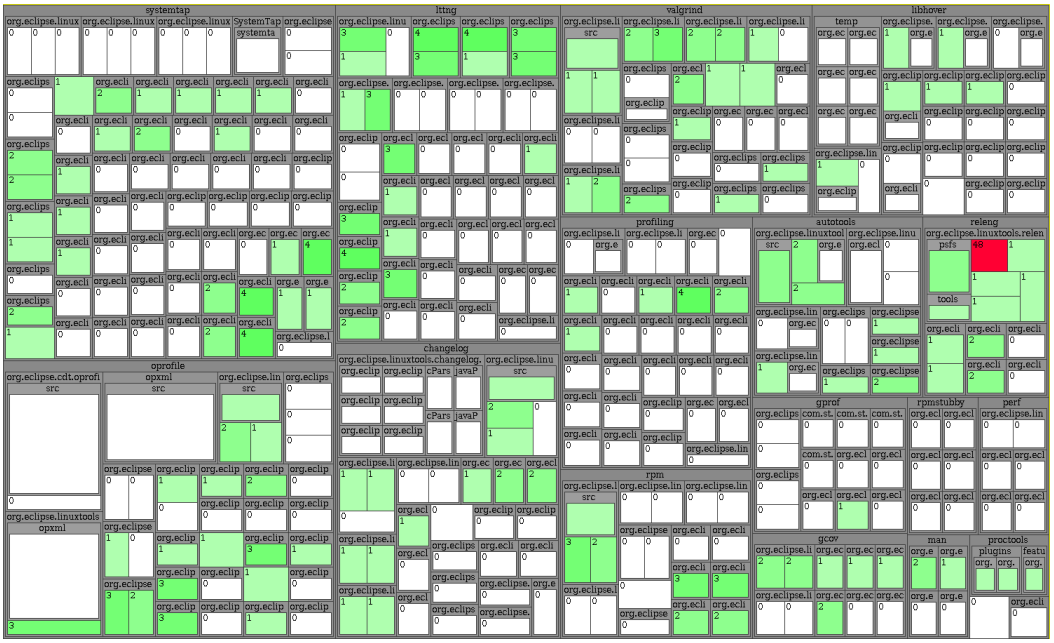

FIGURE 5.66 – Treemap de Ant en 2010

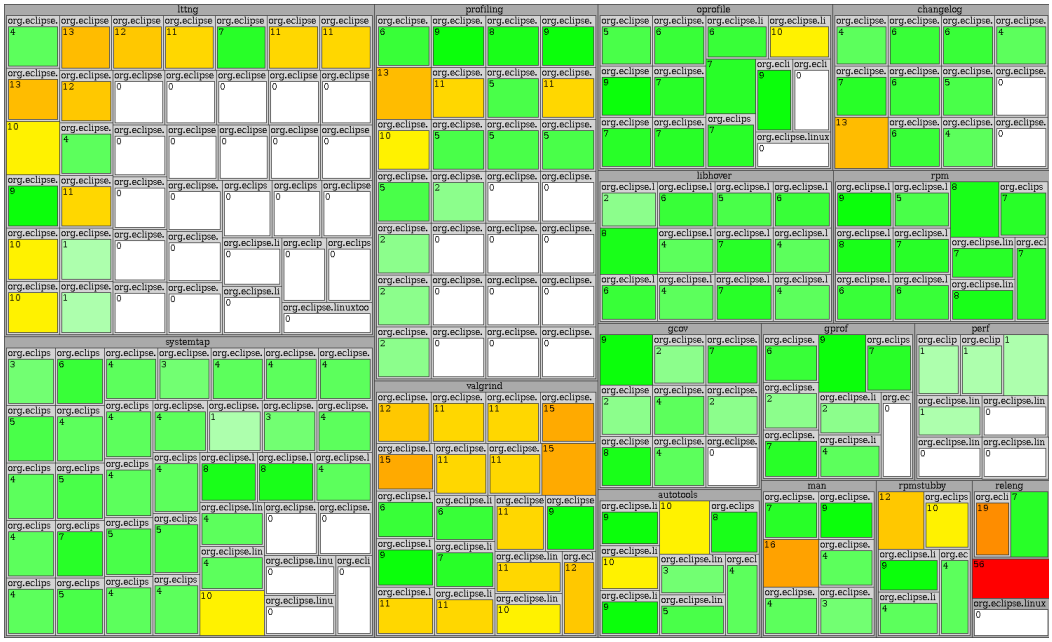

FIGURE 5.67 – Treemap de Maven en 2011

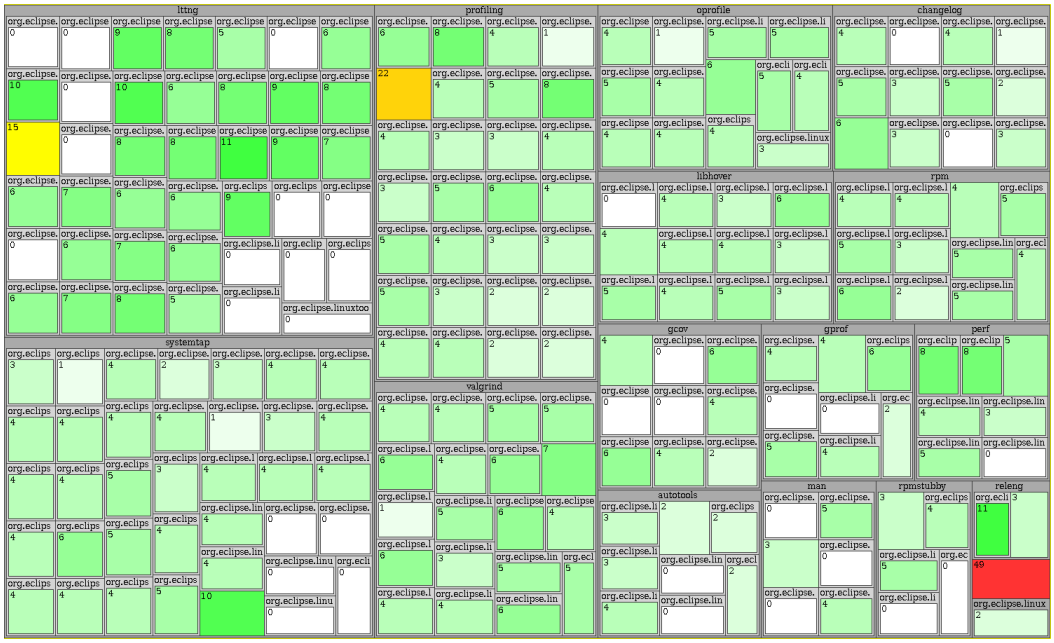

FIGURE 5.68 – Treemap de Maven en 2012

CHAPITRE 6

Évaluation de la méthode

L'approche quantitative permet une bonne estimation de l'effort de travail dédié à la maintenance du Build System, et du gain après la migration. Elle est donc adaptée pour répondre à notre troisième question de recherche, qui vise à connaitre l'impact de la migration.
Cependant, nous avons utilisé les données brutes, sans prendre en compte d'éventuels outils qui pourraient assister les développeurs, en générant par exemple une partie du code pendant la maintenance. Cela peut conduire à quelques imprécisions dans l'évaluation de la charge de travail.
Par exemple, l'analyse manuelle des commits a permis de trouver quelques commits modifiant des fichier pom.xml en regénérant l'intégralité des fichiers au lieu de simplement modifier les lignes nécessaires, ce qui donne un churn énorme alors qu'il aurait pu être de quelques dizaines de lignes.

Par contre, l'approche quantitative n'a pas permis de trouver de tendances claires parmi les projets qui peuvent justifer une migration. Le problème principal est que la diminution de la charge de travail imposée par la maintenance du Build System n'était pas la priorité des développeurs des projets analysés. Leur motivation était plutôt la diminution de la complexité du Build System, qui les empêche de réaliser facilement la maintenance.

Nous pensions que la charge de travail pouvait permettre d'estimer la complexité du Build System, mais le projet BPEL nous a prouvé le contraire. Les résultats montraient un Build System très stable, ne changeant pas souvent avant la migration. Pourtant le développeur de BPEL à qui nous avons posé cette troisième question était celui qui avait le plus insisté sur la très grande complexité de leur Build System Ant.
Notre méthode ne permet donc pas de mesurer conrètement et efficacement la complexité de compréhension et de modification du Build System. Seul le feedback des développeurs nous a permis de d'évaluer la complexité ; sans données quantitatives mais uniquement de façon subjective.

CHAPITRE 7

Conclusion

7.1 Réponses aux questions de recherche

Notre étude nous a permis, à travers l'analyse quantitative et qualitative des sept projets, de répondre à nos trois questions de recherche.

RQ1 *"Quelles sont les raisons qui motivent une migration ?"*

La première question que nous avons posée aux développeurs nous a permis de mieux comprendre les raisons qui les ont motivés à migrer leur Build System.

La raison principale est que Ant est une technologie compliquée à comprendre et à utiliser. De plus, les outils qu'ils utilisent pour gérer leur Build System Ant produisent du code désordonné et de faible qualité. Aussi, la plupart de ces outils ne sont plus activement développé.

Le Build System Ant n'est en plus pas facile à exécuter et à reproduire à chaque fois de la même façon. C'est une barrière pour les contributeurs potentiels qui n'arrivent pas aisément à tester leurs modifications.

Enfin, Eclipse encourage ses projets à migrer leur Build System vers la combinaison Tycho et Maven, pour former un *"Common Build Initiative"*, pour unifier les efforts des développeurs de Build Systems et assurer un meilleur support sur le long terme.

RQ2 *"Quelle est la complexité de la migration ?"*

Les réponses que nous avons obtenu des développeurs nous a permis de répondre à cette question :

La réalisation d'une migration avec Tycho est facile ; de nombreux tutoriels et exemples permettent de comprendre comment réaliser cette migration. De plus, Tycho dispose d'outils permettant de générer tous les fichiers POM, et de détecter efficacement les erreurs.

La réalisation d'une migration avec Tycho est très rapide ; elle peut être réalisée en quelques jours, voire même une seule journée. Avoir de l'expérience dans ce genre de pratiques permet de réaliser les migrations suivantes encore plus rapidement. Les données des projets prouvent cette affirmation ; les migrations se font en un seul ou quelques commits.

Il y a cependant une exception : le projet JBossTools. Les développeurs ont décidé d'utiliser Tycho très rapidement, avant son arrivée à maturité. Ils n'ont donc pas pu bénéficier des outils permettant de faciliter la migration. En réalité, ce sont eux qui ont aidé les développeurs de Tycho à en réaliser. Leur migration a de ce fait été progressive, et beaucoup de parties du Build System Ant précédent ont été conservées pour combler le manque de fonctionnalités de Tycho.

Avec les projets développés en Java, nous sommes donc loin de la situation des projets développés en C comme Linux et KDE. Les migrations que nous avons observées ne sont pas des opérations aussi longues (une année) et difficiles que celles observées par Adams *et al.*

RQ3 *"Dans quelle mesure la migration a eu un impact positif sur la qualité du Build System ?"*

Les effets au niveau quantitatif ne sont pas vraiment significatifs. La réduction de la charge de travail est souvent bien présente, mais elle ne chute pas de façon importante. Par contre l'indépendance du code du Build System par rapport au code source s'est effectivement bien améliorée.
D'autre part, lorsque nous analysons le rapport entre les améliorations quantitatives obtenues après la migration et l'effort, très faible, nécessaire pour réaliser la migration, nous constatons que les gains sont proportionnellement bien réels et justifient la migration.
Le rapport "qualité/efforts" est nettement positif !

D'un point de vue qualitatif, l'amélioration est par contre réellement significative. Beaucoup de développeurs ont témoigné qu'utiliser Tycho et Maven permet d'avoir un code plus propre, plus facile à comprendre, à modifier et à exécuter. C'est un atout important dans le monde open source, car un projet compliqué à construire peut être une contrainte rédhibitoire pour les contributeurs potentiels.

7.2 Travaux Futurs

Il serait intéressant d'étendre la méthodologie pour obtenir un outil complet d'analyse de Build System qui pourrait aider les développeurs à déterminer si leur Build System nécessite réellement une migration.
Pour cela, il faudrait trouver une série de métriques permettant d'évaluer concrètement la complexité du Build System.

Il faudrait aussi étendre l'étude de cas à d'autres projets ayant déjà réalisé ou désirant réaliser une migration du Build System. Ces projets peuvent être d'autres projets open source ne faisant pas partie de l'Ecosystème d'Eclipse et des projets du domaine privé.
L'objectif serait de déceler les tendances quantitatives qui démontrent que le Build System a réellement besoin d'une migration.

Bibliographie

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *Electronic Communications of the ECEASST*, 8, February 2008.

[2] Bram Adams, Roman Suvorov, Meiyappan Nagappan, Ahmed E. Hassan, and Ying Zou. An empirical study of build system migrations in practice : Case studies on kde and the linux kernel. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 160–169, 2012.

[3] G. Kumfert and T. Epperly. The evolution of the linux build system. *Technical Report UCRL-ID-147343*, 2002.

[4] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of ant build systems. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 42–51, Cape Town, South Africa, May 2010.

[5] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of java build systems. volume 17, pages 578–608. Springer, August 2012.

[6] Shane McIntosh, Bram Adams, Yasutaka Kamei, Thanh Nguyen, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 141–150, Waikiki, Honolulu, Hawaii, May 2011.

[7] Alexander Neundorf. Why the kde project switched to cmake and how, 2006.